

AD-A184 127

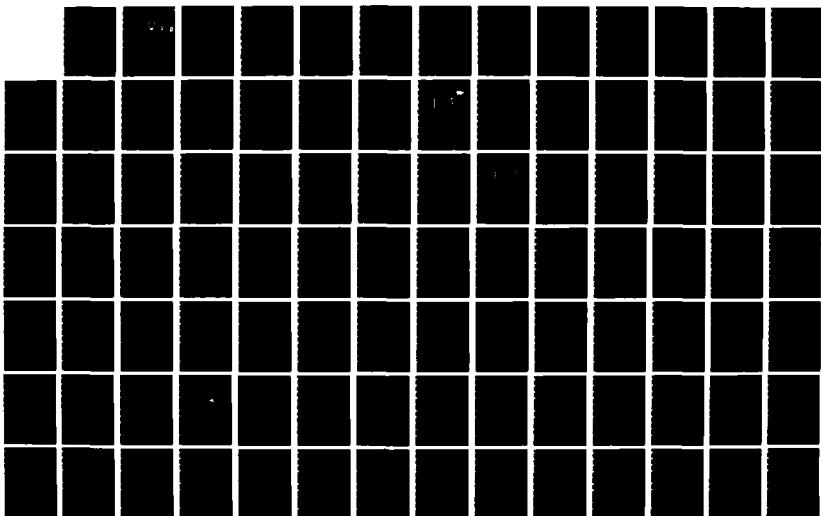
A SURVEY OF OBJECT ORIENTED LANGUAGES IN PROGRAMMING
ENVIRONMENTS(U) NAVAL POSTGRADUATE SCHOOL MONTEREY CA
H HAAKONSEN JUN 87

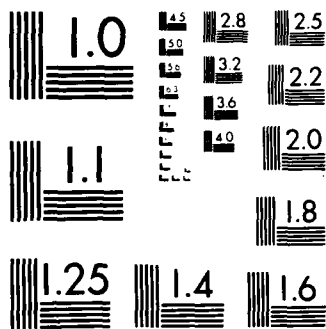
1/2

UNCLASSIFIED

F/G 12/5

NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

AD-A184 127

DTIC FILE COPY

2

NAVAL POSTGRADUATE SCHOOL

Monterey, California



DTIC
ELECTE
SEP 03 1987
S D

THESIS

A SURVEY OF
OBJECT ORIENTED LANGUAGES
IN PROGRAMMING ENVIRONMENTS

by

Harald Haakonsen

June 1987

Thesis Advisor:

Bruce J. MacLennan

Approved for public release; distribution is unlimited.

87 8 28 023

unclassified

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a REPORT SECURITY CLASSIFICATION unclassified			1b RESTRICTIVE MARKINGS		
2a SECURITY CLASSIFICATION AUTHORITY			3 DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited.		
2b DECLASSIFICATION/DOWNGRADING SCHEDULE			5 MONITORING ORGANIZATION REPORT NUMBER(S)		
4 PERFORMING ORGANIZATION REPORT NUMBER(S)			5 MONITORING ORGANIZATION REPORT NUMBER(S)		
6a NAME OF PERFORMING ORGANIZATION Naval Postgraduate School		6b OFFICE SYMBOL (If applicable) 52	7a NAME OF MONITORING ORGANIZATION Naval Postgraduate School		
6c ADDRESS (City, State, and ZIP Code) Monterey, California 93943-5000			7b ADDRESS (City, State, and ZIP Code) Monterey, California 93943-5000		
8a NAME OF FUNDING/SPONSORING ORGANIZATION		8b OFFICE SYMBOL (If applicable)	9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER		
8c ADDRESS (City, State, and ZIP Code)			10 SOURCE OF FUNDING NUMBERS		
			PROGRAM ELEMENT NO	PROJECT NO	TASK NO
			WORK UNIT ACCESSION NO		
11 TITLE (Include Security Classification) A SURVEY OF OBJECT ORIENTED LANGUAGES IN PROGRAMMING ENVIRONMENTS					
12 PERSONAL AUTHOR(S) Haakonsen, Harald					
13a TYPE OF REPORT Master's Thesis		13b TIME COVERED FROM TO		14 DATE OF REPORT (Year Month Day) 1987 June	
15 PAGE COUNT 105					
16 SUPPLEMENTARY NOTATION					
17 COSATI CODES			18 SUBJECT TERMS (Continue on reverse if necessary and identify by block number)		
FIELD	GROUP	SUB-GROUP	object oriented programming; Smalltalk; human-computer interface; interactive integrated programming environment		
19 ABSTRACT (Continue on reverse if necessary and identify by block number)					
<p>This thesis addresses object oriented programming languages; and a restrictive definition of object oriented programming languages is presented and defended. Differences between programming languages are discussed and related to interactive integrated programming environments. Topics related to user friendly interface to the computer system and modern programming practice are discussed. The thesis especially addresses features in object oriented programming languages that are important when a user friendly interactive integrated programming environment is designed. Some future research areas are suggested.</p>					
20 DISTRIBUTION AVAILABILITY OF ABSTRACT <input checked="" type="checkbox"/> UNCLASSIFIED UNLIMITED <input type="checkbox"/> SAME AS RPT <input type="checkbox"/> DTIC USERS			21 ABSTRACT SECURITY CLASSIFICATION unclassified		
22a NAME OF RESPONSIBLE INDIVIDUAL Prof. Bruce J. MacLennan			22b TELEPHONE (Include Area Code) (408) 646-2353		22c OFFICE SYMBOL Code 52M1

DD FORM 1473, 84 MAR

83 APR edition may be used until exhausted

All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE

unclassified

Approved for public release; distribution is unlimited.

A Survey of
Object Oriented Languages
in Programming Environments

by

Harald Haakonsen
Lieutenant Commander, Norwegian Navy
Norwegian Naval Academy, 1977

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

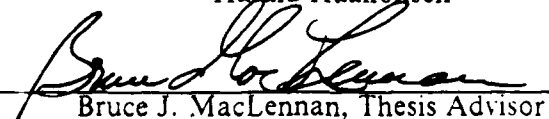
from the

NAVAL POSTGRADUATE SCHOOL
June 1987

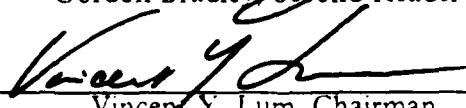
Author:

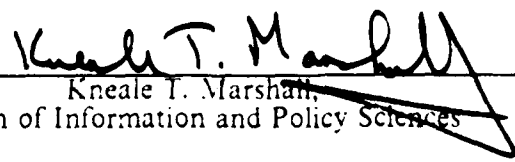

Harald Haakonsen

Approved by:


Bruce J. MacLennan, Thesis Advisor


Gordon Bradley, Second Reader


Vincent Y. Lum, Chairman,
Department of Computer Science


Kneale T. Marshall,
Dean of Information and Policy Sciences

ABSTRACT

This thesis addresses object oriented programming languages; and a restrictive definition of object oriented programming languages is presented and defended. Differences between programming languages are discussed and related to interactive integrated programming environments. Topics related to user friendly interface to the computer system and modern programming practice are discussed. The thesis especially addresses features in object oriented programming languages that are important when a user friendly interactive integrated programming environment is designed. Some future research areas are suggested.

Accession For	
NTIS CRA&I	<input checked="" type="checkbox"/>
DTIC TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
By	
Distribution /	
Availability Codes	
Dist	Availability Codes
A-1	

TABLE OF CONTENTS

I.	INTRODUCTION	11
A.	BACKGROUND	11
B.	OBJECTIVES	11
C.	THE RESEARCH QUESTIONS	11
D.	SCOPE AND ASSUMPTIONS	12
	1. Scope	12
	2. Assumptions	12
E.	LITERATURE REVIEW AND METHODOLOGY	12
	1. Literature Review	12
	2. Methodology	13
F.	SUMMARY OF FINDINGS	13
G.	ORGANIZATION OF THE STUDY	13
II.	INTRODUCTION TO PROGRAMMING LANGUAGES	15
A.	BACKGROUND FOR PROGRAMMING LANGUAGES	15
	1. What is a Programming Language?	15
	2. The Purpose of a Programming Language	16
	3. What are the Criteria for "Good" Programming Languages?	17
B.	INTERFACES IN A PROGRAMMING ENVIRONMENT	18
	1. Interfaces in Programming	18
	2. Dimensions in User Interfaces	20
C.	THE SEARCH FOR A BETTER SOLUTION	20
	1. Procedural versus Nonprocedural Programming Languages	20
	2. What do We Want in the Future	22
D.	WHAT KIND OF HELP CAN HARDWARE OFFER	23
	1. Hardware Cost and Performance, and Its Implications	23
	2. Visual Interfaces	24

	3. Firmware	25
E.	SUMMARY OF THE CHAPTER	25
III.	HUMAN LIMITATIONS AND RELATED TOPICS	26
A.	BACKGROUND	26
B.	HUMAN LIMITATIONS	27
	1. Variations in Performance between Programmers	27
	2. The Cost of Large Complex Software Systems	28
	3. User Interface Performance Issues	28
C.	COGNITIVE SCIENCE	29
	1. Human Memory	29
	2. The Learning Process	30
	3. Thinking and Reasoning	31
D.	WHAT IS "MODERN PROGRAMMING PRACTICE"?	32
	1. Background	32
	2. Modern Programming Practice	32
	3. Why use Modern Programming Practice?	32
E.	WHAT DOES "FRIENDLY" MEAN?	33
	1. User Friendly Interfaces	33
	2. Interactive Systems	34
	3. How do We Learn to use an Interactive System	35
F.	SUMMARY OF CHAPTER	36
IV.	DIFFERENCES BETWEEN LANGUAGES	37
A.	PROCEDURAL VERSUS NONPROCEDURAL	37
B.	PROCEDURAL LANGUAGES	38
	1. Historical Background	38
	2. How do we Cope with the Complexity of Programming?	39
	3. ADA an Example of an Imperative Language	39
	4. Pure LISP an Example of an Applicative Language	40
C.	NONPROCEDURAL LANGUAGES	42
	1. Historical Background	42
	2. Prolog an Example of Logic Oriented Language	43
	3. Object Oriented Languages	44
D.	SUMMARY OF THE CHAPTER	46

V.	WHAT IS AN OBJECT ORIENTED LANGUAGE?	48
A.	HOW TO DESCRIBE AN OBJECT ORIENTED LANGUAGE?	48
1.	General Description	48
2.	Differences between Object and Procedure Oriented Programming	49
B.	TERMINOLOGY USED	51
1.	General Background	51
2.	Objects	53
3.	Messages	54
4.	Classes	54
5.	Instances	55
6.	Methods	55
C.	INFORMATION HIDING	56
1.	Definition	56
2.	Information Hiding in Object Oriented Languages	56
D.	DATA ABSTRACTION	57
1.	Definition	57
2.	Data Abstraction in Object Oriented Languages	58
E.	DYNAMIC BINDING	59
1.	Definition	59
2.	Dynamic Binding in Object Oriented Languages	59
F.	INHERITANCE	60
1.	Definition	60
2.	Inheritance in Object Oriented Languages	60
G.	SOME ADVANTAGES AND DISADVANTAGES IN OBJECT ORIENTED PROGRAMMING	61
1.	Advantages	61
2.	Disadvantages	62
H.	SUMMARY OF THE CHAPTER	62
VI.	INHERITANCE	64
A.	BACKGROUND	64
B.	INTENSION VERSUS EXTENSION	64
C.	INHERITANCE IN GENERAL	65

	1. Inheritance versus Data Abstraction	65
	2. Subclassing	66
	3. Inherited Instance Variable	69
	4. Programmer's View of Inheritance	70
D.	MULTIPLE INHERITANCE	70
	1. Overview	70
	2. Graph Oriented Multiple Inheritance Solution	73
	3. Linear Chain Multiple Inheritance Solution	74
	4. Tree Conversion Multiple Inheritance Solution	75
E.	SUMMARY OF THE CHAPTER	75
VII.	INTERACTIVE PROGRAMMING ENVIRONMENT	77
A.	WHAT IS AN INTERACTIVE PROGRAMMING ENVIRONMENT?	77
	1. Definitions	77
	2. Impact of Tools	78
	3. What is so Special about Programming Environments	79
B.	IDENTITY OF OBJECTS	80
	1. Definition of Identity	80
	2. Identity in Interactive Programming Environments	80
	3. What Language to use in an Interactive Programming Environment	81
	4. Incremental Program Development	82
C.	HOW TO PUT THE USER IN CONTROL	83
D.	LISP IN INTERACTIVE PROGRAMMING ENVIRONMENTS	84
	1. Why use Lisp	84
	2. The Interlisp Programming Environment	84
E.	AN OBJECT ORIENTED INTERACTIVE PROGRAMMING ENVIRONMENT	87
	1. Why use Smalltalk	87
F.	SUMMARY OF CHAPTER	90
VIII.	CONCLUSIONS AND RECOMMENDATIONS	92
A.	CONCLUSIONS	92
B.	RECOMMENDATIONS	93

1. What Can be Done Now	93
2. Future Research Areas	93
APPENDIX A: SMALLTALK-80 TERMINOLOGY	94
APPENDIX B: TOWER-OF-HANOI IN PROLOG	96
APPENDIX C: TOWER-OF-HANOI IN LISP	97
APPENDIX D: TOWER-OF-HANOI IN PASCAL	98
APPENDIX E: TOWER-OF-HANOI IN SMALLTALK-80	99
LIST OF REFERENCES	100
INITIAL DISTRIBUTION LIST	104

LIST OF TABLES

1. EXAMPLE OF PROCEDURAL ORIENTED PROGRAMMING 50
2. EXAMPLE OF OBJECT ORIENTED PROGRAMMING 50

LIST OF FIGURES

2.1	Interfaces in a Programming Environment	19
3.1	Modeling Domains in Programming	31
3.2	User Friendly	34
5.1	Inheritance Example	61
6.1	Without Intersection in Class Membership	67
6.2	Subclasses	68
6.3	Multiple Inheritance	71
6.4	Example of Multiple Inheritance Acyclic Graph	72
6.5	Example of Altered Graph Oriented Multiple Inheritance	73
6.6	Example of Linerized Chain Multiple Inheritance	74
7.1	Incremental Development	82

I. INTRODUCTION

A. BACKGROUND

Traditionally programming languages have evolved towards a higher level of abstraction offered to the programmer. Current programming languages have removed the programmer from the hardware level of the machine, and offered him her increased semantic power of the language which better captures the programmer's concept, but it is not yet normal to work in a fully interactive integrated programming environment.

The problems we want to solve with computers steadily increase in size and complexity. We often talk about "the software crisis," and this can be viewed as a sign that we are reaching the limit of what we currently are able to handle. The complexity barrier is pushed further and further, but we still need to create the software we need in the future to solve these difficult problems. Traditionally the tools in the programming environment have been made by programmers for the benefit of programming. The management's need for tools has therefore not fully been recognized by developers who began the implementation of interactive integrated programming environments.

B. OBJECTIVES

The objective of the thesis is to show some of the aspects that are relevant in the development of interactive integrated programming environments, especially how object oriented programming languages can make our work easier and more efficient.

C. THE RESEARCH QUESTIONS

What is an object oriented programming language, and what is different compared to other programming languages? How can we develop a more user friendly programming environment? What makes it user friendly? What kind of programming environments do we have today? Are any programming languages better than others to build user friendly interactive integrated programming environments?

D. SCOPE AND ASSUMPTIONS

1. Scope

The relevant material for this thesis is vast, so a brutal restriction of the subject has been performed. The discussion is mostly kept on a single user system level in order to reduce the complexity. Typical representatives from each major type of programming languages are studied, and used for comparisons. The sample programs in the appendices are all written in programming languages available on personal computers, e.g. Apple's Macintosh.

2. Assumptions

The thesis assumes that the reader has some basic knowledge of computer science, therefore the commonly used expressions are not defined here. When it comes to the discussion of object oriented programming languages, and inheritance, no background knowledge is assumed. The subjects are covered more in depth with explanations of new concepts. The discussion is based on the relevance to designers and programmers, not so much on management's needs.

The future seems to expose a growing number of people to computer systems, and working environments where computer systems are an integrated part of the whole. Therefore the discussion concentrates on the impact on the average user who is not necessarily a computer specialist.

E. LITERATURE REVIEW AND METHODOLOGY

1. Literature Review

This thesis is a review, combining work from many sources. Most of the literature comes from the academic environment, and each source normally covers only a small research area. Especially when it comes to literature about object oriented programming languages there seems not to be a clearly defined terminology.

2. Methodology

The methodology used is based on an extensive study of literature available in different areas: programming languages, software engineering, cognitive science, computer science, human interfaces, etc. The purpose of the study is to get a feeling for what's involved in interactive integrated programming environments.

Small sample programs of the Tower-Of-Hanoi problem are written in different programming languages (i.e. Prolog, Pascal, Lisp, and Smalltalk) in order to get "hands on" experience, and to better understand the differences between the languages and environments.

F. SUMMARY OF FINDINGS

The thesis concludes that the structure of the language defines the boundaries of the thought of a human being, and that this is valid also for programming languages and programming environments. Object oriented programming languages have four features: information hiding, data abstraction, dynamic binding, and hierarchy of inheritance. This kind of object oriented programming language is well suited for building interactive integrated programming environments that are user friendly. Today interpreted programming languages like Lisp and Smalltalk are the languages that most easily facilitate customizing of an interactive integrated programming environment to the user's needs.

G. ORGANIZATION OF THE STUDY

The first chapters discuss various programming languages, especially the differences between them. The traditional programming languages are not covered in depth, but object oriented languages (Smalltalk) are discussed in more detail. The thesis establishes what minimum criteria a programming language must have in order to be a real object oriented programming language. Some of these criteria are covered in more detail in order to give a better understanding of what object oriented programming languages have to offer the designers' programmers compared to other languages.

Next the comparison is brought a step further, and looks upon the interaction and integration in some sample programming environments. Some of the features in the environments are covered in more depth to bring forward what has been, and still is, important for the evolution of the interactive integrated programming environments. The thesis ends up with conclusions, and some recommendations for future interactive integrated programming environments.

II. INTRODUCTION TO PROGRAMMING LANGUAGES

A. BACKGROUND FOR PROGRAMMING LANGUAGES

1. What is a Programming Language?

The history of programming languages goes back to 1846 when Lady Lovelace programmed Charles Babbage's machine [Ref. 1]. In doing this she showed that she was thinking of a symbolic system as a language. Many would say that it was her knowledge of mathematics, and not her knowledge of poetry, that led her to this abstraction. Mathematical entities are abstractions that do not change over time, and the mathematical theories are well accepted and understood.

Computer programming is a complex human activity, and the programming language is the tool used to get the hardware to do what the programmer wants done. Programming languages have changed over time. In the beginning it was machine language, but as the complexity of the tasks we wanted the computer to solve for us grew, we got symbolic assemblers, higher level languages, and symbol manipulation languages. We have pushed the complexity barrier further and further as human beings have tried to understand, and write, programs an order of magnitude larger than what has been feasible previously.

The focal point in the computer science problem solving process is still the programming language and the programming environment. Features in the programming language can affect the way a programmer approaches the design of a solution to a particular problem. A linguistic theory, the Sapir-Whorf hypothesis [Ref. 2], states that the structure of a language defines the boundaries of the thought of a human being. There is a strong interaction between languages and thought. The structure a language presents for manipulating words and the vocabulary available for representing ideas constrain the thoughts that can be easily and accurately represented. In addition the structures and patterns that characterize people's thought process affect how they are able to use the facilities provided by a programming language. In other words, a limited programming language will be a handicap for the programmer who tries to realize his/her full problem solving potential, so that he/she must improvise to get an acceptable solution to the problem he/she wants to solve.

The enormous variations in symbols, constructs, and syntax observed among natural languages is also true of computer programming languages. Computer programming language differences range from the long compound words of Cobol to the symbolic brevity of APL, from the massive size of Ada to the compactness of Pascal.

2. The Purpose of a Programming Language

Programming languages are used to write programs in order to get some computer hardware to perform a useful function. These programs have a dual function, communication between human and machine, but also human to human communication. A programming language must therefore provide all the necessary interfaces with the hardware of the computer system; at the same it time must also be able to capture the ideas of the programmer. High levels of abstraction increase the semantic power of the programming language, and capture better the problem solving concepts of the programmer. Programming is not a branch of mathematics; it is a unique form of communication in which human beings take an active role and machines often a passive role.

The programming language, with its structure, can help us define the boundaries of our thoughts. We can tailor a language to suit our special needs, such as APL for mathematics or special database languages (query languages) for large collections of data, in order to reduce the distance between the user and the way the user thinks about the problem.

One way of classifying programming languages is by the extent to which they force one to write in machine level procedures, rather than in natural languages. This scale runs from machine languages and assembly languages, through high level programming languages, through query languages, to natural language. As one moves up the scale, the structures in the programming language take over more and more of the details of integrating the software with the hardware. Programming languages at the upper end of the scale make computers more accessible to more people, mainly because they are not forced to understand the hardware in order to interact with the computer system.

3. What are the Criteria for "Good" Programming Languages?

A perfect programming language must be ideally suited for all situations, for all users, for all applications, and for all computer systems. Today the programming language, in addition, must be unambiguous, because the current state of technology has problems letting the context decide the accurate meaning of the statements.

To design a language that is easy to understand and use, and at the same time is powerful, we must accept a tradeoff between the principles such as the following from MacLennan's list [Ref. 3: p. 526-527]:

1. Abstraction: Avoid requiring something to be stated more than once; factor out the recurring pattern.
2. Automation: Automate mechanical, tedious, or error prone activities.
3. Defense in depth: Have a series of defenses so that if an error is not caught by one, it will probably be caught by another.
4. Information hiding: The language should permit modules designed so that (1) the user has all of the information needed to use the module correctly, and nothing more; (2) the implementor has all of the information needed to implement the module correctly, and nothing more.
5. Labeling: Avoid arbitrary sequences more than a few items long; do not require the user to know the absolute position of an item in a list. Instead, associate a meaningful label with each item and allow the items to occur in any order.
6. Localized cost: Users should pay for what they use; avoid distributed cost.
7. Manifest interface: All interfaces should be apparent (manifest) in the syntax.
8. Orthogonality: Independent functions should be controlled by independent mechanisms.
9. Portability: Avoid features or facilities that are dependent on a particular machine or a small class of machines.
10. Preservation of information: The language should allow the representation of information that the user might know and that the compiler or interpreter might need.
11. Regularity: Regular rules, without exceptions, are easier to learn, use, describe, and implement.
12. Security: No program that violates the definition of the language, or its own intended structure, should escape detection.

13. Simplicity: A language should be as simple as possible. There should be a minimum number of concepts with simple rules for their combination.
14. Structure: The static structure of the program should correspond in a simple way with the dynamic structure of the corresponding computations.
15. Syntactic consistency: Similar things should look similar; different things should look different.
16. Zero-one-infinity: The only reasonable numbers are zero, one, and infinity.

It is at the present time not possible to design a programming language that is ideal in all situations, for all users, for all computer systems, for all applications, even if we do have the principles as in the above list. Today's programming languages are specialized for specific areas of use, in order to keep the size and complexity within the human being's limit. Some high level programming languages, like PL. I and Ada, are designed to cover a wide area of applications, but the cost is increased size and complexity.

B. INTERFACES IN A PROGRAMMING ENVIRONMENT

1. Interfaces in Programming

The environment in which a programmer performs his/her task includes: the physical environment, the presence or absence of other people, the personalities of the other members of the group, directives from management, learned programming methodologies, reference manuals for the programming language and the computer system. All of these affect the programmer in his/her job. The term "programming environment" can also be used more specifically, namely for a set of computerized tools which ease the communication between the human being and the computer system [Ref. 4: p. 559]. There are several interfaces in a programming environment that are important for determining how the human being thinks and reacts [Ref. 4: p. 142]:

1. Between the user's conceptualization of the actual world he/she wants to represent and the programming language in which the user must describe this world so that the computer system can simulate it.
2. Between the programming language and the visual presentation of the language to the user.

3. Between the visual presentation of the language and the way the user must physically indicate what action should take place.

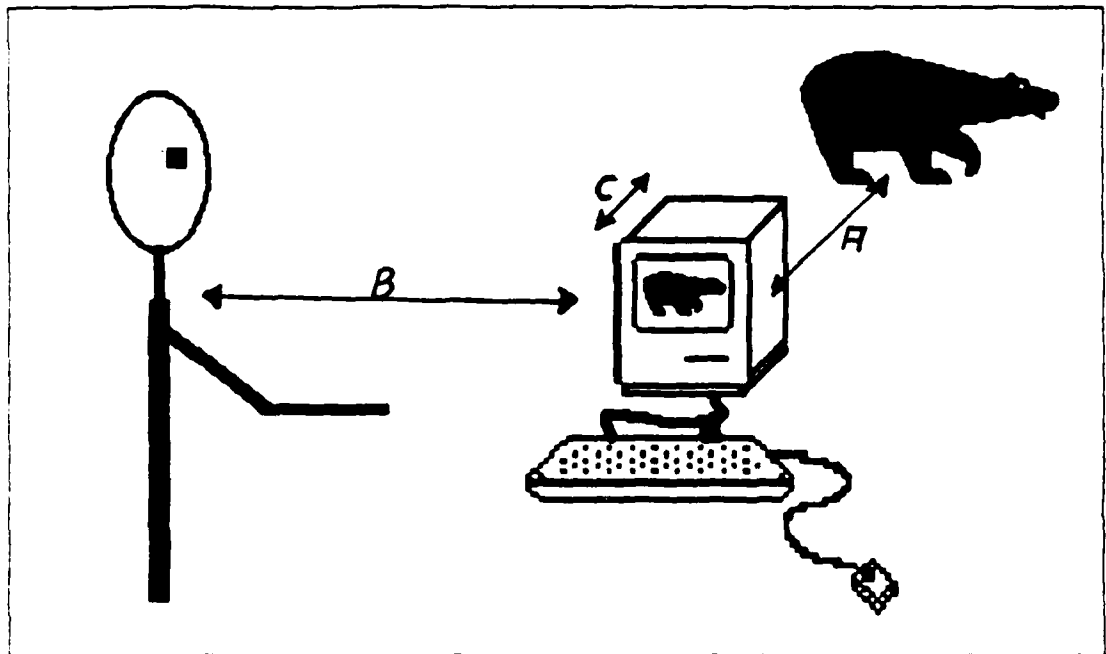


Figure 2.1 Interfaces in a Programming Environment.

These three interfaces are visualized in Figure 2.1 where: A is the real polar bear to the simulated polar bear, B is the user interface to the computer, and C is the simulation described in the computer.

The features of a programming language are the working tools for the programmer. The programmer's work in finding a solution is affected by the tools available to him/her. The interfaces are part of this, and they lead the programmer towards certain problem solving methodologies, but it might be that it is not the best methodology for this specific situation. People normally spend more time describing data manipulation than they do describing control flow [Ref. 5: p. 184-215]. Traditional programming languages on the other hand provide for the development of large control structures with embedded data manipulation. The natural human tendency seems to be to start with data manipulation and add control flow as an afterthought. Miller [Ref. 5] concludes that natural language is not adequate for procedural specifications, but that a limited and structured subset of natural languages might be more effective, and make the human-computer interface more friendly.

2. Dimensions in User Interfaces

The programming environment must be accessible to the human being in order to be of any use. The interface between the human and the system is complex, and consists of a large number of design decisions. The most important interfaces are listed next, with some relevant questions added [Ref. 6: p. 13]:

1. Presentation; how are objects displayed and selected? Does the system translate structure into text, i.e. pretty printing? Can it give more than one representation of the same object, i.e. multiple views?
2. Command interface; how are commands invoked? Are the menus context sensitive or not? Select then command, or command then select? Binding commands to keys?
3. Extensibility; how can a user tailor the system to his/her needs? Does the system support extensions? What is the performance of extensions? What kind of mechanisms are used for the extensions?
4. Window systems; what is the underlying technology for implementing the interface? What is accepted in the windows? How does the system support the windows? Is the window system compatible with other systems?

The list is only meant to give an idea of the complexity involved when designing a user interface. The question of how to present a programming environment to the user is not yet fully solved. Both the technology and the methodology are immature, and we have many contenders in the field. A very good example, of new ideas in user interfaces, is the Macintosh personal computer produced by Apple Computer Inc. This system shows how it is possible to give a novice user access to a powerful personal computer system. The Macintosh is in many ways an interactive environment that lets the user stay in control throughout the session. All applications written for the Macintosh system are supposed to follow a standard (defined by Apple) user interface in order to reduce the learning time for the user. It is of course possible to violate these standards if one wants to.

C. THE SEARCH FOR A BETTER SOLUTION

1. Procedural versus Nonprocedural Programming Languages

In the early days of computers, a few decades ago, the job for the designers and programmers was to convert manually existing systems to new technologies using computers. Today we still convert and refine existing systems, but increasingly the job

is to apply technology to do something new, things we never dreamed the previous system could do. When we are not replacing an existing system, design no longer follows directly from analysis. Analysis of what exists yields insufficient information to design what will come to exist, so analysis and design become inseparable. The problems we seek to solve using computer systems become more and more complex. The traditional, obvious applications for computer systems, have already been done and newer applications are often:

1. More complex.
2. Less obvious.
3. Larger.
4. Used for longer period of time.
5. More likely to change over time.

This problem complexity, and the necessity to deal with many different details at one time, makes programming generally very difficult. We are trying to master this complexity by applying what has worked for us in the past. The programming language designers have realized this, and have given us temporary relief, but every time the problems we want to solve outgrow the current programming technique. Symbolic assemblers, higher level languages, and symbol manipulation languages have in turn pushed the complexity barrier back [Ref. 4]. Programming language design is a cumulative learning process, and programming is still a very young branch of engineering. The evolution of programming languages has resulted in solutions to a broader class of problems, and even new approaches toward the solution of presently unsolved problems.

We have different considerations that dictate the design of programming languages [Ref. 3: p. 523]:

1. Uses (problems solved).
2. Users.
3. Computers on which the programming language can be implemented.
4. Successes and failures of the designs of the past.

These different considerations show that it is very difficult today to construct a single language that can cover all possible needs, even if we have the well defined design principles stated earlier in this chapter.

The search for a better programming language has given us a wide variety of different languages, and dialects of languages. A nonprocedural programming

language is one that lets the programmer concentrate on "what" he she wants the program to do, instead of "how" to do it. Related to this is the separation of the logic component and the control component within the language. Examples are Prolog and Smalltalk, and this thesis will cover them in more detail later. Procedural programming languages are the more conventional languages, in which the programmer has full responsibility for the control component. Examples are Ada and Lisp, which also will be covered in more detail later.

The terms procedural and nonprocedural will be discussed in more detail later. No well defined and agreed upon definition exists, but examples from different types of programming languages will be used to clarify the differences.

The procedural languages are often divided into two subclasses, Imperative and Applicative languages. Imperative programming languages includes most of the traditional languages (eg. Fortran, Cobol, Pascal), but not languages like Lisp, Prolog, and Smalltalk. Prolog and Smalltalk are not applicative languages either. Imperative languages depend heavily on an assignment statement and a changeable memory for accomplishing a programming task. Most of these languages are basically a collection of mechanisms for routing control from one assignment to another. In an applicative language on the other hand the central idea is function application, that is to apply a function to its argument. A subset of Lisp can be used as an applicative language.

2. What do We Want in the Future

Historically the introduction of high level programming languages relieved the designer programmer from the machine code by introducing higher levels of abstractions. The future should give us high level programming environments that provide help for the designer/programmer in understanding and manipulating complex software systems. The human user should not worry about the detailed specification of algorithms, but rather work with the description of the properties of the packages and objects we use to build programs. The programming environment should give us a higher level of abstraction so that we can specify behavior, i.e. what to do instead of how to do it.

D. WHAT KIND OF HELP CAN HARDWARE OFFER

1. Hardware Cost and Performance, and Its Implications

The economics of data processing are changing rapidly. Historically, the hardware cost of a computer system was so high that concern with hardware efficiency was not only justified, but essential. Therefore programmers worried about the CPU time and memory space their program code needed. Today the declining cost of hardware makes developing and maintaining of many programs more expensive than running them. Therefore emphasis is shifting from efficiency on the computer to controlling software cost and user friendliness. Software cost in this context is the total accumulated cost over the whole life cycle of the application, i.e. problem definition, specification, design, coding and testing, implementation, maintenance, and purging. The shift in cost also affect the design of programming languages, and at the moment we have a wide variety of experimental languages taking advantage of the increased performance of computer hardware.

Alan Kay [Ref. 3: p. 453] in the late 1960s was convinced that in the future it would be possible to put the power of what was then a room sized, million dollar computer into a small machine (personal computer) placed on a person's desk. He asked himself what kind of language would be needed for this machine, and decided that a simulation and graphics oriented programming language could make the computer power accessible to nonspecialists. Xerox Corporation started design of Smalltalk based on his ideas long before suitable machines were around. Xerox developed Smalltalk as a software system, rather than creating a specific hardware package. The experience gained by developing applications in one Smalltalk system was used to generate next generation of Smalltalk, and so on. The current system, Smalltalk-80, was developed to be adaptable for implementation on a large number, and variety, of computer systems. This is one of the few cases where the language is ready before the hardware to run it on.

Many of the facets of programming are currently caused by the way we "adjust" the user to the hardware available. The nature of programming is going to change in the future, as the computer technology matures. Current programming techniques are not adequate for building and maintaining systems of the complexity called for by the tasks we attempt to solve. In the future we need to shift our attention away from the detailed specification of algorithms, towards the descriptions of the properties of the modules and objects with which we build programs. Already today

the memory available is so large and so cheap (relatively) that we can see changes in the way people program. Higher efficiency of the hardware also removes much of the work previously spent on speeding up modules using assembly language. Today higher level languages can be used without much worry about efficiency problems because the optimized compilers have become very good.

Today we are not very good at reusing old designs and modules (code). The wheel is reinvented many times over because our current programming languages and methodologies do not enforce reuse as a resource saving method. Some of the problems are caused by the lack of knowledge of existing modules that can be reused. How do we build libraries etc.? Reuse of code is a complex and not well understood problem today. There are many ways to attack the problem, and this thesis looks at what object oriented programming languages can offer to reduce the problem of reusability. The topic of reusability will not be covered in depth, but also hardware development makes a difference. Because of a better performance to cost ratio of computer hardware it is today feasible to build libraries of general routines that can be used in more applications with minor changes. General routines are normally slower than optimized specialized routines, but because a general routine can be used more often it is also possible to put in more resources to optimize it.

2. Visual Interfaces

The human interface has come a long way since the introduction of computers. In the beginning even the assembly language (mnemonics) was thought of as very "human" and "user friendly," but today this is viewed as primitive. Today's bitmapped high resolution screen, with a pointing device (normally a mouse), gives the user quite a different interface to the computer system. The growing capabilities and performance of hardware are used to ease the interaction between the human being and his/her hardware. Alan Kay's dream from the 1960's is a reality today. We do have personal computers with the power of yesterdays mainframes, and new programming languages like Smalltalk use simulation and graphic presentation to make the system accessible to the nonspecialist [Ref. 7]. Systems like Macintosh and Amiga are examples of how this has become available to the consumer who has little or no background in use of computers.

3. Firmware

Traditionally a system could be described by its software and its hardware. Today this is not quite true because the difference between software and hardware has become rather fuzzy. Everything that can be done in software can be implemented in hardware and vice versa (not quite true because we need some hardware components to run the software on). The introduction of a user friendly interface put a much heavier load on the computer resources, and firmware has therefore been implemented to speed up execution. A typical example is the Macintosh. When it was presented in 1984 it had 128k in RAM and 64k in ROM. The ROM is an example of firmware that contains very efficient routines that control the user interface, and at the same time is accessible to application programs.

E. SUMMARY OF THE CHAPTER

Computer programming is a complex human activity, and the programming language is the tool used to get the hardware to do what the programmer wants done. Programming can be viewed as a unique form of communication in which human beings take an active role and machines often a passive role. The structure of a programming language defines the boundaries of the thought of human beings, i.e. the programming language limits our ability to solve problems.

A perfect programming language must be ideally suited for all situations, for all users, for all applications, and for all computer systems. At the present time no such perfect programming language exists, but we have more specialized languages that keep the complexity within the human being's limit.

III. HUMAN LIMITATIONS AND RELATED TOPICS

A. BACKGROUND

More and more people get involved in tasks where a computer is used. It is no longer only a small group of specialists that perform design and programming. Already today a large number of people use specialized application languages, like Lotus 1-2-3, in their daily job situation. Most people do not think of using a word processing package as programming, but in reality it is programming at a very high level of abstraction for a very specialized context. In the future people will have routine daily interaction with computer systems. How do we build such systems? What limitations does the human being have when it comes to the use of computer systems? This is the kind of questions we now, and in the future, must answer. This chapter will try to give an overview of some of the features that are involved. In addition some ideas from David Lorge Parnas are discussed in order to show that not only the programming languages are important for which problems we are able to solve, but also how we actually use the languages.

"Cognitive Science," "Cognitive Psychology," and "Human Information Processing" all help provide the conceptual framework needed to think about the abilities and limitations of the person designing or using the computer system. There are three basic factors involved when a computer system is designed:

1. Know your user: experience, limitations, ability, and motivation.
2. Know your user's task: visual and manual, what must be done.
3. Know your user's working condition: where the job is being done, what it is like there.

All the three factors important to the system design involve the person that is supposed to use the production system. The user of the system seems to be the limiting factor for many implementations of computer systems. The human being has many disadvantages, i.e. we forget, we get tired, etc. On the other hand we also have many advantages over computer systems (at least today), i.e. we are good at recognizing patterns, and at setting a situation into the correct context.

B. HUMAN LIMITATIONS

1. Variations in Performance between Programmers

No commonly accepted theory of significant factors in programming or program design seems to exist, but there have been some studies of programmer performance. Brooks [Ref. 8: p. 737-751] found enormous variations in performance between different programmers of comparable experience for the same programming task. He estimated factors between 5 and 100, and suggested that differences in the strategy used by different programmers caused these large variations.

Perhaps programming activity is too complex a human behaviour to be studied in detail, and must therefore largely remain a mysterious process. The only way to achieve the necessary knowledge about the programming activity is to systematically study programming behavior. Experiments have both dependent and independent variables. Dependent variables are what you measure, and must be selected to capture the part of the programming task you are interested in. If you use several dependent variables to measure more aspects of the performance, the sum total of the information (e.g. time used, numbers of errors, design strategy, rated ease of use) will give a better picture of what's going on. An independent variable can be the menu length, from which the appropriate choice must be selected. This kind of variable can be used to measure programmer performance. Human knowledge is by necessity incomplete. We cannot know in advance what we might be able to know and what might be essentially unknowable [Ref. 9]. We must try to find things out, or as Einstein said: "The important thing is not to stop questioning."

Programmer limitations affect how much coding is lost because the programmer did not have full mastery of his/her computer, programming language (and environment) or himself/herself, or a combination of these. In addition the programmer may be unaware of a certain algorithm, or unable to grasp a sufficiently large portion of the problem at one time to get the overall picture of the problem he/she wants to solve. The problem we are facing today is how to give the programmer full mastery of his/her job, and at the same time give him/her the powerful tools needed to solve the large and complex problems we currently want to find a solution to.

2. The Cost of Large Complex Software Systems

We have long advocated economy of scale for manufacturing, but does this apply to software systems? To produce a large integrated software system takes a long time, and costs large amounts of resources. If we are totally dependent upon bug free software, we are at the moment in deep trouble. The testing dilemma was stated by Dijkstra [Ref. 10: p. 6] as: "program testing can be used to show the presence of bugs, but never to show their absence." Today program testing is possibly the only way to ensure an acceptable quality of a complex program. Although a large integrated software system is more economical when it is in active use, it naturally costs more if we must take it out of use due to an error. Because it is larger, there are also more things that can go wrong, so things will go wrong more often. Finally because of its size it is more difficult to find out what is wrong, and it will therefore take a longer time before it is running again. Human limitations, when it comes to complexity of systems, seem to reduce the advantage of large integrated systems. Especially if the modules depend upon each other, and therefore if there is an error in one, none can be used. Some advocate that systems like the "Strategic Defence Initiative" (SDI) are outside our capabilities when it comes to software design because of its size and complexity. David Lorge Parnas strongly opposes the SDI due to human limitations; he said: "the state of the art in software is significantly behind that in other areas of engineering." [Ref. 11: p. 1327] The lack of testability of SDI software makes life difficult; he stated: "we would need a software system so well-developed that we could have extremely high confidence that the system would work correctly when called upon." [Ref. 11: p. 1328]

The bottom line is that not only do large integrated systems today cost an enormous amount, but we also have the "cost" related to trust in the system, i.e. do we trust our systems enough to let them control our life? The cost related to trust is more related to politics than money, therefore it is less measurable and much more complicated to agree upon.

3. User Interface Performance Issues

Vision is our primary sense, but it has its limitations. There are many things the eye does not see, so the screen on the terminal computer can take advantage of this. The human visual system has several characteristics that control the use of computer systems for visual representation of data and information. The human

perceives as simultaneous changes that occur in less than 20 ms. And if successive frames on a screen of a moving image are redrawn in less than 50 ms, the human perceives the object as moving smoothly. And finally if feedback to user initiated events are produced in less than 300 ms, the human perceives it as occurring instantaneously. These indicated speeds represent limits for which objects move smoothly and do not interrupt the user's train of thought. The bottom line is that the human is very slow compared to computers, so the drain on computer resources to give a good visual presentation is little compared to the advantages it can give. [Ref. 12: p. 176]

Colors used on the screen also do have an impact on how we react [Ref. 13: p. 3-33]. Because vision is our primary sense the designers of interactive programming environments must take this into account. Use of colors in the interface often means that the resolution is poorer than with a monochrome screen, i.e. there is no such thing as a "free lunch."

C. COGNITIVE SCIENCE

1. Human Memory

Andrew Monk [Ref. 13: p. 49] states: "Human memory is currently believed to be a complex system of independent storage systems." These storage systems have different characteristics. Long term memory seems to use a semantic code and stores information in a highly organized manner. No capacity limit seems to exist for this storage. Retrieving information seems to be a process of reconstruction, instead of just output of held information. That means that the output may be different from what was put in. The short term memory is easier to measure, test and is therefore better understood. It consists of "buffers" that holds the information for a very short period of time while it is processed by the brain and stored in long term memory. The long term memory seems to be limited to only being able to code information that is meaningful to the user. Therefore language designers and program designers must ensure that the user has a mental model of the system used. The system environment should work the way human beings think. Current programming languages often are designed to ease the parsing problem instead, i.e. the use of prefix constructs instead of infix constructs.

2. The Learning Process

Learning is a complex process of integrating new knowledge into a structure of what is already known. Writing a program is a process of learning, both for the programmer and the person who is going to use the software product. This learning takes place in a context of a particular hardware system, particular programming language or programming environment, and depends also on the society around us. Much of the information the maintenance programmer has, is in the source code of the program. This is a typical example of the dual role of a programming language, i.e. human to human, and human to computer system, communication.

Reading and understanding a computer program listing is a cognitive task that is critical to the development and maintenance of software. In the interaction with the computer system the user often receives and sends information in the medium of written language. This information, the words and sentences used, must be related to each other for the reader to be able to understand the information as a whole. The reader is performing a problem solving exercise [Ref. 13: p. 37]. How the human understands this piece of written information depends upon the characteristics of the text, but also it depends on the programmer's past experiments and familiarity with the concepts involved. Especially in the maintenance phase, because normally this person is not the person who wrote the program in the first place, it is necessary to understand how the program works in order to update or enhance it.

The construction of a program can be viewed as building a mapping from some domain external to the program into the set of objects and operations available in a particular programming language or system. The task of understanding the program becomes one of constructing information about the modeling domains used to bridge between the problem and the executing program. Within each domain there is information about the basic sets of objects, including their properties and relationships, the set of operations that can be performed on the objects, and the order in which these operations take place [Ref. 14]. Figure 3.1 gives a visual representation of a "banking" problem domain.

Natural languages may be the only language that can serve across multiple domains. The problem is that natural languages are inconcise and overload many words and expressions. Perhaps we ultimately have to change our ways of thinking a little bit in order to fit our computers, because current technology does not allow natural language as the communication between the human and the machine.

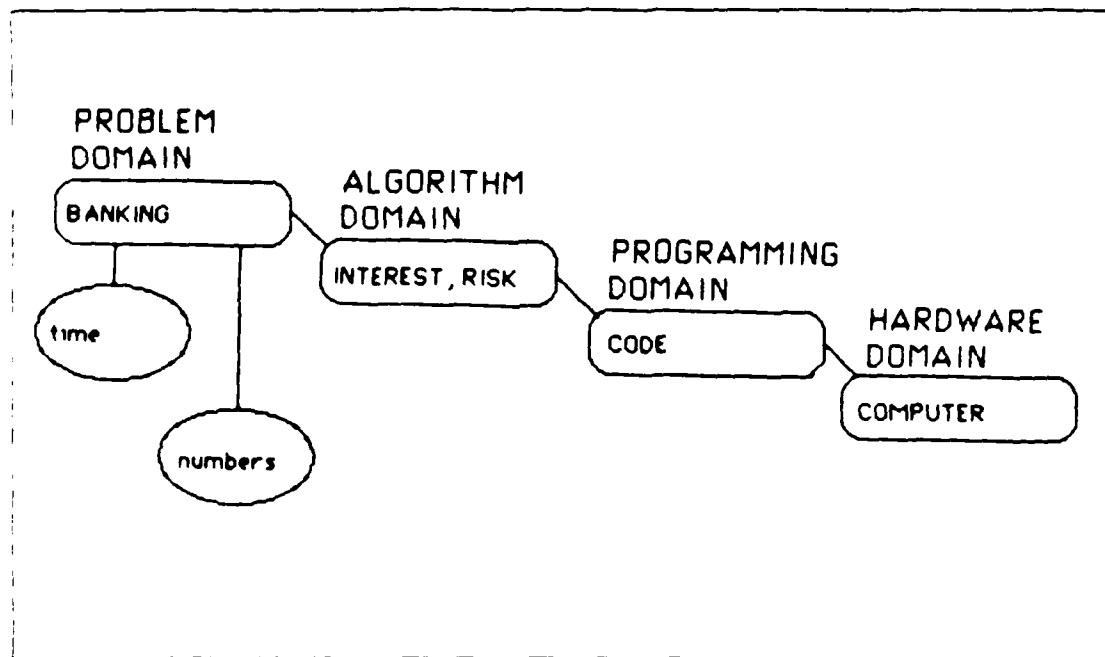


Figure 3.1 Modeling Domains in Programming.

3. Thinking and Reasoning

Neil Thomson [Ref. 13: p. 5] describes the human being: "Of all the human facilities, our ability to reason is the one which appears to set us apart from the animals." Thinking seems to be a combination of inductive and deductive reasoning, but at the same time the two are difficult to separate. The process of arriving at a logically necessary conclusion from initial premises is deductive reasoning, and it is a logical process. Inductive reasoning on the other hand is not a logical process, but refers to the production of a general statement from specific instances. The problem is, how can one build computer systems with a reasoning process as flexible as human reasoning. So far we have not been able to do this, but we have a large number of expert systems that are used as working tools in specialized domains. Typically the expert system asks questions and the user supplies the expert system with the facts needed. One problem with expert systems is to specify and quantify the knowledge of an expert in a specialized field; another problem is how to write the program that uses this knowledge in an intelligent way.

D. WHAT IS "MODERN PROGRAMMING PRACTICE"?

1. Background

The "software crisis" has not been solved yet. New languages have moved the complexity barrier further and further, but not really solved the problem "how to develop large and complex programs." David Parnas [Ref. 10] says we must learn to use the programming languages we have. The problem is not so much the coding itself, as the analysis and design of the programs in order to make the coding a straightforward process. In other words we must understand the problem and our programming language in order to be able to solve new, large, and complex problems.

2. Modern Programming Practice

Modern programming practice is helpful in solving complex problems. This is a programming technique where more work is put into the analysis and design phase in order to save later on in coding, maintenance, and enhancements. David Parnas is a strong supporter of this methodology, which emphasizes the following:

1. Structured programming.
2. Information hiding.
3. Data abstraction.
4. Top down design.

The basic principle of structured programming is like the military strategy of divide and conquer. A program, or software module, is broken into small, independent, single function modules which are clear and easy to follow logically. These modules are themselves composed of even smaller blocks: the decision, sequence, and repetition structures found in most programs. The details of structured programming will not be covered in depth because they are of little relevance to this thesis, but the other elements of the methodology will be covered in more or less detail in the context of the programming languages and programming environments discussed.

3. Why use Modern Programming Practice?

Many of the problems we are working on today are large and complex. A large number of people and resources are involved, therefore we have a need for management of the project. The management is responsible for: allocation of resources, deadlines, go or no go decisions, funding etc. To be able to take the decisions, and make comparisons between different projects, it needs measurables. In this context measurables are anything that can be measured. Measurables are

quantified data that is used for comparisons, evaluation, etc. In a production environment the management is normally the one who specifies the policy, standards and interactive integrated programming environment to use. Many of the tools in the environment will therefore be used for managing the whole lifecycle, and not only for the design code phase. There may be a conflict of interest between management and other users of the system because of the measurables. Often technical people do not want to give measurables because they are afraid of later being confronted with them.

It is better to be clear than right, because if you are clear someone might tell you that you are wrong. The system must be managable, i.e. measureable for the management. In addition, maintenance of software is expensive and time consuming. The best way to minimize the maintenance cost is to design the program with ease of maintenance in mind. One way to do this is to use modern programming practice, and one key is information hiding. If this is done, most changes will be limited to the logic contained in a single module, and the possible number of interactions between the modules decreases enormously. Therefore the ripple effect will be minimized, as changes made to an independent module should have little or no impact on the logic performed by other modules. Modern programming practice often results in levels of abstractions, levels of protection, i.e. programs that have a layered structure.

E. WHAT DOES "FRIENDLY" MEAN?

1. User Friendly Interfaces

The environment in which programmers work is a rich and complex environment, full of human involvement and interaction, change, and misleading appearances. In this context friendliness would be the distance between the things the user thinks about doing, and the things the user actually can do in the programming environment. The three interfaces, see Figure 2.1, can give a measurement of how friendly the system is to a user.

People are different, and react differently to a given stimulus. This means that what is friendly to one person may not be equally friendly to another person. Generally a visual representation, along with reduced interdependencies among different parts of the system, makes a programming environment more friendly.

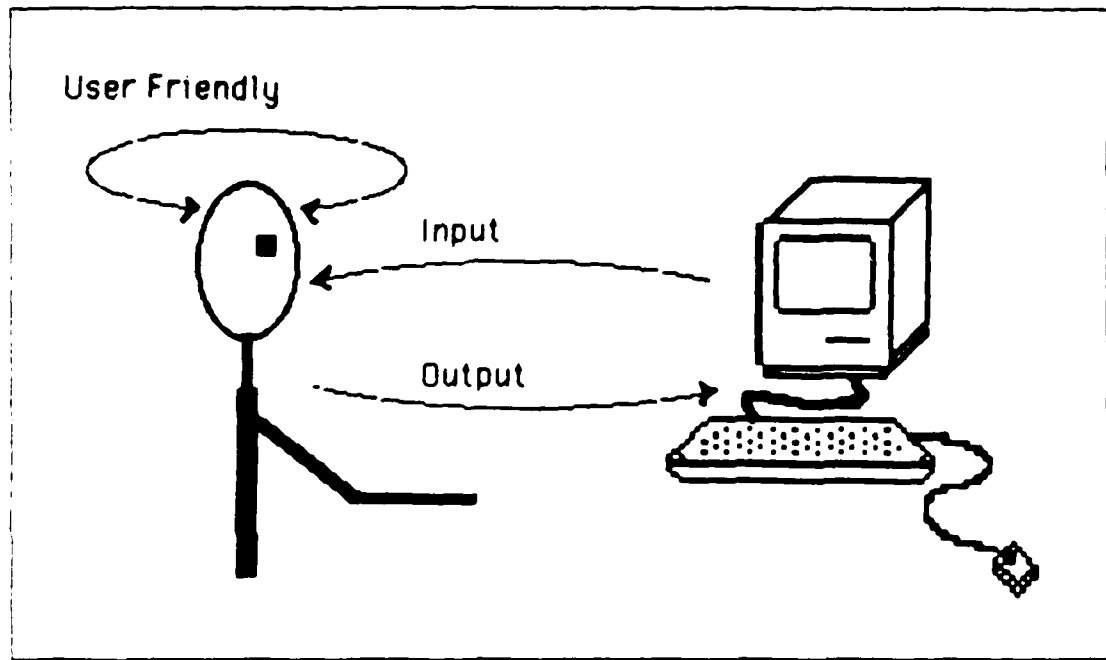


Figure 3.2 User Friendly.

User friendliness depends on how easily the human can process input from a computer system, and how easy the human can output something to the computer system and have it understood. See Figure 3.2 for a visual representation of user friendliness. Note that we are talking about the input to the human being and not (as it normally is represented) to the computer system. This is done to emphasise the shift in design that are taking place in some of the more modern implementations of user interfaces, which put the user in control throughout the session. The interaction with the system is on the user's premises. The trend is to concentrate more and more on user friendly interfaces between the user and the computer system. On many systems this shows up as windows, menus, and use of a mouse. The objective is to make the process of communication between the user and his/her computer system more intuitive, i.e. more natural to the user.

2. Interactive Systems

A wide variety of hardware can be used in an interactive environment in order to help communicate information from the user to the computer system: keyboard, mouse, pointing device, speech etc. At the same time there are many ways the computer system can communicate informations to the user. This communication of

information must be made such that it is easy for the user to find the information required, as well as to make it easy to understand once found. An interactive integrated programming environment should offer the user many different structural views of the program. Examples are: when editing the programmer may want to view a program as a parse tree, when reading as a document (text), when debugging as a control flow graph, and when looking at the screen as a visual image. Computer resources should be used to make this interaction as intuitive as possible to the user, i.e. the user should not have to fight the system.

3. How do We Learn to use an Interactive System

Today interactive integrated programming environments, mostly based on Lisp, seem to be characterized by tools that interact with each other frequently. The programmer may easily switch between an editor, interpreter, debugger, and compiler. Each of these tools must know, to some degree, about other tools. The user interface must be as natural as possible for the human programmer. To achieve this the environment must take advantage of all the human's strong sides: ability to recognize patterns, visual sense, auditory sense, etc.

We have a great deal of knowledge about the human visual and auditory senses. In addition we have more or less accepted theories about how we think etc. But at the moment we do not know how to put all this knowledge together in order to produce the optimum interactive integrated computer system. The human being is characterized by the ability to change context without loss of information. We are often working on one problem, but suddenly we get an idea or maybe the thought just wanders off, and we find ourselves in another context. If the system we are working with could support also this human activity it would be easier to learn and operate. This means that the system has to be modeless to the user. A consistent user interface can help the user overcome some of the problems he/she meets when being exposed to a new interactive system, or a new application within an already known interactive system. The system should let the user be in complete control throughout the session, and offer the user the structural view he/she wants to use.

F. SUMMARY OF CHAPTER

Today more and more people are exposed to computers in their daily life. Therefore more computer resources must be spent on the interface between the user and the computer system in order to reduce the learning problem for the average user.

Our ability to solve large and complex problems is not only limited by our programming languages, but also by the way we think about the problem. Modern programming practice, which emphasizes: structured programming, information hiding, data abstraction and top down design, can help us use what we currently have to solve problems we earlier could not handle.

Currently we do not know how to put all available information together in order to produce the optimum interactive integrated computer system. The user interface must be as natural as possible for the human user.

IV. DIFFERENCES BETWEEN LANGUAGES

A. PROCEDURAL VERSUS NONPROCEDURAL

The terminology and names used in computer science are not always clearly defined and nonambiguous; this is true for procedural and nonprocedural programming languages also. In this thesis the nonprocedural languages are the ones that emphasise the features that let the programmer concentrate on "what" he/she wants the program to do, instead of spending time and effort to tell it "how" to do it [Ref. 3: p. 499]. The separation of the logic components and the control components within the language is also important to nonprocedural languages [Ref. 15: p. 424-436]. The terms procedural and nonprocedural are relative terms, and many computer scientists would say "more procedural" and "less procedural" instead.

In logic programming languages like Prolog, the goal directed use of Horn clauses lets the programmer express the facts, and removes the "how" part of the problem. Prolog does have control features built in that the programmer can use, such as the "cut" predicate. Prolog will be covered in more detail later.

Object oriented programming languages are in this thesis classified as nonprocedural programming languages. This claim will be defended later in the thesis. The discussion is more on programming style than the programming languages themselves because it is possible to program in a procedural style in a nonprocedural programming language (i.e. Smalltalk). In a conventional procedural language the programs are active and the data elements are passive. The program elements act on the data elements. In an object oriented programming language, like Smalltalk, the data elements are active and respond to messages that make them act on themselves. Maybe they modify themselves, or maybe modify or return other objects. Object oriented languages will be covered in more detail later.

The control structure of procedural languages determines the order in which actions take place within the program. The statements within the program must be executed in the specified order to ensure correctness. This implies a very close relationship between the control structures and the actual logic of the program.

Theoretically all programming languages are equally powerful because they all build on machine code, and they all can represent a Turing machine. For human beings this seems not to be true, but this is only because the programming languages do not equally well support the way people think, and often different languages are inappropriately called "less powerful" than some other language.

B. PROCEDURAL LANGUAGES

1. Historical Background

Traditional programming languages do not really offer the programmer a high level of abstraction, but rely on a "word at a time" programming style [Ref. 4: p. 404]. The von Neumann architecture of the computer system, and the sequential nature of the traditional programming languages place limitations upon the level of abstraction the language designer can make available to the programmer. Programming has proved to be much easier if a specialized language, for building similar applications, is available to the programmer. The language supplies appropriate abstractions for the application domain, and a programmer can simply select and compose these high level constructs to build an application. The primitive constructs in the language are then application level constructs. These specialized languages can be successful in a specialized problem domain, but are generally less useful for other applications, and therefore these application modules are difficult to reuse. For a programming language to take advantage of both specialization and generality, it is possible to build the specialized language upon a more general base language, which must be extensible.

Special languages like the symbolic language APL can be very compact, with very high semantic power. If the same program were produced with Pascal, a much larger program (in number of codelines) would be the result. All languages have the same theoretical power, and therefore the same problem can be solved in all the programming languages. APL is less procedural than Pascal, less has to be said about "how" and this makes APL more compact.

The evolution of these languages, and especially the effort to provide more semantic power, has resulted in the development of Ada. This very large and complex language provides increased semantic power (higher abstraction) at the cost of simplicity, clarity of understanding, and maybe the programmer's mastery of his her

tools [Ref. 16]. Lisp is a language with a long history, and is very popular in research and development communities because everything is treated as lists. Therefore it is easy to modify and extend to suit special requirements. The programmer is also in this case responsible for the control structure of the program.

2. How do we Cope with the Complexity of Programming?

The complexity of programming increases dramatically with the size of the program, because the possible number of interactions between the modules that make up the program increases so enormously. How can we deal with this problem? What has worked best in dealing with unmastered complexity in the past is a combination of the following [Ref. 17: p. 8]:

1. Learning from analogous situations outside the present situation.
2. Learning how people think and combine that thinking with facts and preconceptions to determine actions.

Traditionally courses in computer programming teach you to think like a computer when designing a program. This view of programming does not support information hiding, which is one of the critical ideas in "modern programming practice." In procedural languages the programmer is responsible for the control structure of the program, and this might be the reason for "think like a computer" programming. Ada is an example of a language that is designed to incorporate the new programming ideas, but still allows the programmer think like a computer if he/she wants to.

Our ability to grasp a complex problem is controlled by short and long term memory. Short term memory is a workspace of limited capacity that holds and processes those items of information occupying one's attention. The capacity of short-term memory is according to Miller [Ref. 18] seven plus or minus two chunks. A chunk is a single symbol or a group that has a single meaning, i.e. telephone number, piece of code etc. As programmers mature they observe more algorithmic patterns and build larger chunks. The scope of the concepts that the programmers have been able to build into chunks provides one indication of their programming ability, and it is rather easy to measure.

3. ADA an Example of an Imperative Language

Imperative languages depend heavily on an assignment statement and a changeable memory for accomplishing a programming task [Ref. 3: p. 344]. Most of these languages are basically a collection of mechanisms for routing control from one assignment statement to another assignment statement.

The United States Department of Defence (DoD) initiated the development of Ada for the purpose of large computer programs to be used in embedded computer systems. Ada is a rich and complicated language, but no subset is allowed by DoD. Since Ada is designed as an integrated, unified language it is difficult to remove features without disturbing the unity of the remainder of the language. The language supports "modern programming practice," and is a block structured language suitable for general purpose programming. Even if Ada has taken many ideas from Pascal, it is not Pascal, and the design style of programs will be very different. Pascal is a language for single, independent programs, while Ada is a language for designing and creating large software systems. It is designed to help create programs by piecing together standard program units from an Ada library. Information hiding is supported by modularization (package), and this helps simplify readability, maintenance, and enhancements. Abstraction both for data and control is achieved with the features in Ada.

The dependencies among program units are explicit in Ada source code. The structure of an Ada system is not dispersed among the source code, a run time executive, and auxiliary system generation tools. The structure is completely defined by the source code. The Ada compiler ensures that units are compiled in a correct order, and that an executable system is configured only from units that are up to date.

Some have expressed concern about Ada, that it may be too large for the average programmer to learn completely, and that the programmer therefore will lose mastery of his/her working tool. Hoare in his 1980 Turing Award Lecture was critical of the size and complexity of Ada. He meant that the language contained too many features that made it complicated to use effectively. Much of the complexity comes from the wide variety of features to handle the same problem. It is not like Lisp where everything is a list, even the program itself, or like Smalltalk where everything is an object. The complexity of the features in Ada may make it more complicated to build user friendly interactive integrated programming environments.

4. Pure LISP an Example of an Applicative Language

In an applicative language the central idea is function application, that is, applying a function to its argument [Ref. 3: p. 345]. Within an applicative programming language, functions may be defined explicitly, conditionally, recursively, or as the composition of other functions. The main point is that these functions

operate only on data (numbers, characters etc.). Functions that take other functions as arguments provide a higher level of abstraction.

Lisp is one of the oldest languages around, second only to Fortran. It was developed in the late 1950s out of a need for artificial intelligence programming. In this kind of application the interrelationship of data and information must be represented. The result is that the pointer, and in particular linked list structures are natural data structuring methods [Ref. 3: p. 341]. Standard Lisp is not a pure applicative language since it, among other things, has an assignment operation: "setq." To get a pure applicative language, a subset of Lisp can be used, namely, Lisp without "setq," "rplaca," and "rplacd," but also with "eq" restricted to atoms.

An applicative subset of Lisp encourages higher levels of abstractions than conventional Lisp, because the assignments are hidden from the user. The assignments may well exist on a lower level, but they are hidden from the higher levels of abstractions. An applicative programming language makes it easier to integrate different tools into a system because the problem of side effects is avoided. MacLennan [Ref. 19], among others, have suggested combination of features from value oriented (applicative) programming and object oriented programming in order to build interactive integrated programming environments.

Lisp is a symbol manipulation language, where the fundamental objects are called atoms. Groups of atoms form lists. An atom is a sequence of alphanumeric characters; a list is a sequence of atoms and other lists, enclosed in parentheses. Lists themselves can be grouped together to form higher levels lists. The ability to form hierarchical groups (e.g. lists of lists) is of fundamental importance to the language [Ref. 20: p. 2]. Atoms and lists collectively are called symbolic expressions. Symbol manipulation is often called list processing.

Lisp represents both the data and programs by the use of lists. This uniform way to represent everything in a program makes Lisp ideally suited for interactive integrated programming environments, because everything can be treated as lists. A symbolic manipulation program uses symbolic expressions to remember and work with data and procedures.

Lisp and Smalltalk both perform dynamic type checking as opposed to the static type checking in Pascal and Ada. Both Lisp and Smalltalk will be discussed more in detail later, especially how they are used in interactive integrated programming environments. We will see that dynamic binding today makes it easier to build and maintain interactive integrated programming environments.

The sample program Tower-Of-Hanoi in Appendix C is written in Lisp [Ref. 20: p. 88-90].

C. NONPROCEDURAL LANGUAGES

1. Historical Background

A pure nonprocedural programming language would be one in which the programmer only had to state what was to be accomplished, and leave it to the computer system to determine how it was to be accomplished. Normally this kind of programming is achieved using a higher level of abstraction that removes the user from the trivial part of specifying control flow etc. Prolog, which stands for "programming in logic," was developed in France [Ref. 3: p. 500] in 1972. Since then a number of interpreters and compilers have been developed for a number of different computer systems. Prolog is becoming more and more popular for logic programming, and some even think it will replace Lisp for artificial intelligence programming in the long run. It was chosen as the language the Japanese will use on their 5'th generation supercomputers. Although there are many logic programming languages, this thesis will investigate Prolog since it is a typical, and well known, representative of logic programming languages.

Object oriented languages like Smalltalk are another typical but less known, example of nonprocedural languages. Many computer scientists believe that object oriented programming is the only possible way to go if we want a reliable implementation of a very large and complicated software system (like SDI). This is mainly because we, at the moment, are not able to perform a convincing test of conventional written programs. Exhaustive testing is not possible due to the enormous number of possible interactions between modules. The only way today to prove correctness, i.e. the absence of bugs in the program, is by exhaustive testing. In object oriented programming the testing problem is reduced to testing the objects. Everything is treated as objects, it is therefore easy to add to or change a program. The class structure of an object oriented language, like Smalltalk, prevents objects from making too many assumptions about the internal behavior of other objects.

2. Prolog an Example of Logic Oriented Language

Prolog builds on first order predicate logic [Ref. 21: p. 14]. Computer languages try to do things, but logic just says that certain things are true or false. Prolog looks like logic. That is, its syntax is that of logic, but the semantics are different. In logic programming, programs are expressed in the form of propositions that assert the existence of the desired result. The theorem prover then must construct the desired result in order to prove its existence. Thus a side effect of the proof produces the wanted result.

Programming in Prolog is different in style from most programming in other languages, and it is called declarative programming. The main idea is to write programs as lots of small modular pieces. The programmer's emphasis is on writing correct pieces, and not on figuring out how the pieces will go together. The emphasis is on whether each piece makes sense by itself and whether it is logically correct, not on what it does. Prolog makes the "closed world assumption" or "lack of knowledge" inference [Ref. 21: p. 71]. That is, if you cannot prove something is true, assume it is false.

Prolog is not a pure nonprocedural language because it has built in some special features. The "cut" predicate for example cannot be interpreted declaratively, and has no argument. It always succeeds when you first encounter it. As a side effect it throws away backtracking information to prevent normal Prolog backtracking. A cut predicate at the end of a rule means you want to solve the problem only once.

As opposed to Prolog, pure logic programming allows separation of logic and control. Because the clauses of a logic program have no effect upon the correctness of the program, the meaning of the program is tied to the logical relationship of the program clauses, not to the order in which they are executed [Ref. 3]. The programmer can therefore focus on the details of the logic component when he/she is concerned with program correctness. After a correct program has been established, the programmer can concentrate on the control component for efficiency considerations. Appendix B shows a small sample program written in Prolog where also the cut predicate is used.

A frequent danger in Prolog programming is infinite loops. Typically a rule calls itself forever, or a set of rules call one another in a cycle. This happens more often in Prolog than in other languages because of the emphasis on recursion and complicated backtracking [Ref. 21: p. 43]. Artificial intelligence programs tend to be

large, complicated, and very hard to debug. A problem suitable for artificial intelligence techniques must be sufficiently well defined to have clear criteria for success and failure. Otherwise it is impossible to tell if it is working or not. Prolog generally has problems when processing time, storage space and calculation accuracy are critical. In addition most people find it difficult to read and understand large Prolog applications.

Pure logic programming has some disadvantages that result from the absence of state changes. Therefore it is difficult to write programs for those applications for which temporal change is an essential element. Examples are: databases, graphics, real time programs, interactive systems and operating systems. Many of the components in an interactive programming environments are of the above type: editors, debuggers, version control, management systems etc. But Prolog, as opposed to pure logic programming, has "assert" and "retract" to take care of state changes, but this is not enough to make Prolog well suited for implementing interactive integrated programming environments.

3. Object Oriented Languages

What is meant by "object oriented" is still rather vague, and a lot of confusion seems to exist even today among computer scientists. Some call Lisp an object oriented language, others claim that Ada is object oriented [Ref. 22: p. 11-19]. There is a trend to give existing programming languages an extension that includes object oriented features but the question is if this makes them real object oriented languages. An object oriented programming language is one in which the fundamental processing paradigm is simulation. In such a language one often sends a message to an object, rather than the more traditional approach of calling an active procedure to operate on some passive data that is passed to it. Object oriented programming qualitatively enhances the design, creation and maintainability of software systems. Much of this power derives from modularity, and the fact that absolutely everything can be handled as objects. The message is not a distinguishing factor in object oriented programming languages, but is often incorrectly used to describe them.

The object oriented process is closer to high level application programming, and further from the machine level, than the traditional approach. The object oriented approach allow design and coding to be done at the same time. There may be an object in the library that nearly fits the requirements, and you can start the coding with this object. Instead of creating new modules out of smaller subparts, you create new objects by modifying existing objects. This means that instead of piecing together, you

carve and shape an existing object. This kind of design code technique seem, for many, to be more natural and intuitive than the more traditional design code process.

Object oriented programming languages as they are defined in this thesis are nonprocedural because the programmers do less of the "how" than in more traditional programming languages. The classification is based on the style of programming because object oriented programming languages can be "misused", i.e. used in a procedural manner.

Modularity is at the moment probably the best technique available for managing the complexity found in software systems. This means that certain types of complexity are strictly contained within boundaries called module interfaces. Many programming language environments include features that support modularity, but only above the level of the procedure call. The object oriented approach incorporates modularity at the most basic level of a software system.

A restaurant can be used as an example to demonstrate the difference between the object oriented and the traditional approach. In an object oriented restaurant, you order your food by sending the chef a message (via the waiter). The chef is assumed to have the prerequisite knowledge to take the order from there and prepare the food, but you may also give the chef a specific recipe if you want to. In a procedure oriented restaurant, you must send the recipe to the chef. This means that you must know something about cooking in order to get a meal, and may even need to know something about the chef. The procedure specification in the procedure call establishment puts constraints on the chef. Suppose, unknown to the patron, the chef had learned a better way to prepare the meal. Only in the object oriented restaurant could the new skills of the chef really be exploited to the benefit of everyone.

What are the minimum requirements to be an object oriented language? What features must be supported? This thesis will try to show that the language must support the following four features to be a real object oriented programming language:

1. Information hiding.
2. Data abstraction.
3. Dynamic binding.
4. Inheritance hierarchy.

Especially the last feature, inheritance hierarchy, reduces the number of languages that can be called object oriented. This feature also suggest that this thesis support the simulation paradigm, more than just message passing, as the most important

description of object oriented languages. This definition of object oriented programming languages excludes, among others, the following languages: Ada supports packages (untyped data abstraction) but not inheritance, CLU supports clusters (typed data abstraction) but not inheritance, standard Lisp does not support information hiding, and ISO Pascal does not support information hiding.

There are other concepts in object oriented programming that are not central to the basic idea of object oriented programming. One is automatic storage management, which is not necessary, but is very useful when implemented. Automatic storage management techniques such as garbage collection and reference counting let programmers ignore details concerning the release of an object's storage in memory. Typically garbage collection in a real time system is solved by one of the following methods:

1. Sweep garbage collection.
2. Separate memory management processor.
3. Parallel.
4. Split virtual memory in two; cost is only one bit in address.

These techniques make the application source code cleaner, and the overall software system more reliable. Another concept is the virtual memory system needed to take advantage of all the classes (objects) created. Standard Smalltalk does not have a virtual memory capability, but this problem does not exist in most of the newer implementations of object oriented languages. The standard Smalltalk-80's object format allow a simple resident implementation, but at the same time facilitate easy extensions to virtual memory. In the future it may be feasible to swap objects instead of pages in such a system. LOOM [Ref. 23: p. 251-270] is an example of an experimental single user virtual memory system that swaps objects and operates without assistance from the programmer.

D. SUMMARY OF THE CHAPTER

The programming languages can in general be classified as procedural or nonprocedural. In procedural programming languages the programmer is responsible for the control structure of the program, while in nonprocedural programming languages the programmer concentrate on what he she wants the program to do

instead of how to do it. Ada and Lisp are examples of procedural programming languages, while Prolog and Smalltalk are examples of nonprocedural programming languages. Theoretically all programming languages are equally powerful, and can represent the Turing machine.

V. WHAT IS AN OBJECT ORIENTED LANGUAGE?

A. HOW TO DESCRIBE AN OBJECT ORIENTED LANGUAGE?

1. General Description

Most people view processing in an object oriented system, like Smalltalk, as simulation [Refs. 19,24]. The programming language objects correspond to real world objects, and the manipulation of these objects are simulated by sending messages to the programming language objects. Simulation is particularly appropriate to systems that must deal explicitly with the passage of time and the alterations of objects through time. Examples of these systems are: interactive systems, graphics systems, operating systems, editors, file systems, version control systems and database systems. Smalltalk and Simula are built upon the framework of a conventional procedural language, and this framework does not take full advantage of the simulation paradigm. Some people feel the message sending is the key to describe object oriented languages [Ref. 25], but this thesis shows that this is not enough because the four features described in the last chapter are not guaranteed by message sending alone.

Many people who have no background in how computers work find the idea of object oriented systems quite natural. This is probably caused by the close correspondence between thinking about computer objects and the real world objects. Often the object oriented program is derivable from the real world situation it is intended to model. In Smalltalk they concentrated on the visual impact of bitmapped high resolution graphics, on highly interactive user interfaces, and on increased support for the user in the design and programming role. The enhancement to the visual interface covers the basic concepts of windows, menus, and scrollbars. In addition the interaction between the user and the system emphasized the use of a pointing device (mouse) rather than keyboard for selecting objects, and operations on objects. The basic idea about how to create a software system in an object oriented fashion comes more naturally to those without a preconception about the nature of software systems [Ref. 26: p. 74].

Object oriented programming is a technique well suited for organizing very large and complex programs. It makes it practical with current available technology to deal with problems that otherwise would be impossibly complex. An object oriented program consists of a set of objects and a set of operations on these objects. The definitions of the operations are distributed among the various objects that they can operate on, i.e. the system is not monolithic. At the same time, the definitions of the objects are distributed among the various facets of their behavior [Ref. 27: p. 1]. The data values inside an object can represent the properties and relations in which that object participates, and the behavior of the programming language object can model the behavior of the corresponding real world object. The main paradigm of object oriented programming supported in this thesis is simulation, which is quite natural when you know that many of the ideas in object oriented languages are taken from Simula (Simula = simulation) [Ref. 3: p. 464]. The creators of Smalltalk at Xerox see the language characterized by the following three principal attributes [Ref. 23: p. 10]:

1. Data stored as objects which are automatically deallocated.
2. Processing effected by sending messages to objects.
3. Behavior of objects described in classes.

An object oriented language is organized around objects. These objects are places for data storage, like Pascal records. In addition they have methods, which are routines that operate on the object's data. In object oriented programming you decide on your data structure first, and then afterwards you decide what routines you need to operate on the data structures. You can do this in all languages, but in object oriented languages you can group the data structures and the routines together into objects. An object is like a little program, that does its task in an independent manner. Each program "knows" how to do its task, like the chef in the example described earlier knows how to prepare the meal you are ordering.

2 Differences between Object and Procedure Oriented Programming

The restaurant chef example mentioned earlier gives a good feel for the difference, but the following examples will demonstrate more of the organizational difference between the two types of programming we are discussing, procedure oriented and object oriented. Most existing programs are procedure oriented, but nonprocedural languages like Lotus 1-2-3, Focus, DBIII etc. are now becoming more and more important in our normal job situation. Many of these new application specific, nonprocedural languages, sacrifice efficiency for a more user friendly interface

In a procedure oriented program the programs are organized around procedures and functions, and the programmer decides what task needs to be done.

TABLE 1
EXAMPLE OF PROCEDURAL ORIENTED PROGRAMMING

```

staff_member = RECORD
professor    = RECORD
student      = RECORD
PROCEDURE Salary(personel)
  IF personel = staff_member THEN calculate salary this way
  ELSEIF personel = professor THEN calculate salary another way
  ELSEIF personel = student   THEN calculate salary third way
END
PROCEDURE Exercise(personel)
  IF personel = staff_member THEN exercise at this time
  ELSEIF personel = professor THEN exercise at another time
  ELSEIF personel = student   THEN exercise yet another time
END

```

TABLE 2
EXAMPLE OF OBJECT ORIENTED PROGRAMMING

```

staff_member = OBJECT
  PROCEDURE Salary calculated this way
  PROCEDURE Exercise at this time
END
professor    = OBJECT
  PROCEDURE Salary calculated another way
  PROCEDURE Exercise at another time
END
student      = OBJECT
  PROCEDURE Salary calculated third way
  PROCEDURE Exercise at yet another time
END

```

For the examples assume a program that operates a school, in our case operates on staff members, professors and students. The program need to implement payroll (salary) and physical fitness (exercise) for the staff member, professor and student. A very high level of pseudo code is used in the following examples just to highlight the interesting points. The examples are incomplete programs (i.e. the declaration part) just intended as an illustration of the organizational difference between procedural and object oriented programming. Both examples are supposed to solve the problem without any debate of which is the best methodology to use.

The difference between a procedure oriented program and an object oriented program is a matter of style. Both approaches can do exactly the same things, but each

approach has advantages in certain programming areas. The advantages of object oriented programming result from the simulation paradigm; in particular, they are well suited for programs that deal with time, and changes of state in time.

There are several ways to look at the difference between object oriented programming and procedure oriented programming [Ref. 28: p. 147]:

1. Code viewpoint, in terms of the program structure that is created in the program. Each object that is created can be viewed as an independent entity in the program. Each object operates on the data passing through it according to its own built in rules. Changing the object's methods means changes to the built in rules.
2. Data viewpoint, in terms of the data structures the object handles. Each object not only stores information (data), but also processes the data, i.e. each of the objects operates on the information within itself.
3. Structural viewpoint, in terms of the resulting way to design code the program. Instead of piecing together smaller modules to larger ones, you specialize modify existing objects.

B. TERMINOLOGY USED

1. General Background

The term "object oriented programming" was first used to describe Smalltalk programming environments developed at Xerox. Smalltalk took many of its most important ideas, such as classes and objects, from a simulation language called Simula that was based on Algol-60, and designed in Norway in the 1960s. The two languages are different in a number of ways. Simula-67 contains Algol-60 as a subset, and supports: block structure, static (lexical) name binding, and compile time type checking. Smalltalk has none of these features; it is more in the style of Lisp with uniform representation, dynamic binding, and run time type checking. In Smalltalk the designers combined the incremental program execution of Lisp with Simula's class and virtual concepts. Simula was designed from the beginning as both "a system description language and a simulation programming language." [Ref. 29: p. 128]

This thesis will use object oriented programming concepts, terminology and characteristics from Xerox Palo Alto Research Center. The designers of Smalltalk decided to let everything in the system be an object. This was not only applied to the basic datatypes, but also extended to the state of the system: activation records,

instructions, and program counters all followed a specified format. The return address for every subroutine call and even the program counter is an integer offset, not an absolute address. Everything was constrained by this design, even the most frequently accessed of all data, i.e. the instructions. Smalltalk is defined in terms of an interpreter for a virtual machine with a set of instructions. "Smalltalk-80: The Language and Its Implementation," by Adele Goldberg and David Robson plays an important role in this picture, and is a de facto standard when it comes to concepts and terminology for object oriented programming. The description of Smalltalk in this thesis builds on work done by Goldberg and Robson. See appendix A for terminology for Smalltalk-80.

The Smalltalk-80 programming system is divided into two major components: the Virtual Machine and the Virtual Image. Protection of the software is done by copyrighting the Virtual Image [Ref. 23: p. 4]. The modular design of Smalltalk makes this approach for protection feasible. The virtual machine for a particular computer system consists of an interpreter, a storage manager, and primitives for handling the input output devices. The virtual image is a large collection of objects that make up descriptions of classes providing basic data structures, basic graphics and text, viewing and user interface support, compiler, decompiler, and debugger. Because Smalltalk is defined in terms of an interpreter, the virtual machine is easy to implement. All systems running the Smalltalk-80 programming system would therefore look the same to the user; each system supports bitmapped graphics and a pointing device. The research effort in Smalltalk environments focuses on increasing the support that the computer system can provide to users without a background in computers. The research is centered on the visual impact of bitmapped graphics, on highly interactive user interfaces, and on increased flexibility in terms of user programmability.

These design decisions show that the human interface is given priority over hardware considerations. Therefore Smalltalk executes rather slow due to, among others, the following reasons:

1. Smalltalk is defined in terms of an interpreter, and interpreters are slow.
2. Smalltalk is uniformly object oriented and this implies a large number of messages (procedure calls), which are time consuming.
3. Smalltalk creates and destroys a large number of objects. The memory management system therefore has a lot to do.

There are five major concepts in Smalltalk: objects, messages, classes, instances, and methods [Ref. 30: p. 6-16]. The Smalltalk language is based on these

five consistent abstractions. System components are represented by objects. Objects are instances of classes. Objects interact by sending messages. Messages cause methods to be executed. Like Lisp, Smalltalk seeks to provide uniform treatment of different kinds of information: text, graphics, symbols, and numbers. By packaging the behavior of each form of the information with the actual data, the information can be shared between programs without changing representation.

2. Objects

Instead of two types of entity that represent information and its manipulation independently, as in procedure oriented languages, an object oriented language like Smalltalk has a single type of entity, the object that represents both. In a programming system that is uniformly object oriented, like Smalltalk, a class is an object itself.

An object is an instance of a class, and represents a component of an object oriented system. The objects represent the components of a software system. Objects may have a number of relationships with other objects. "One object may be part of another object, or (as in an operating system) the owner of another object." [Ref. 31: p. 6] An object consists of some private memory and a set of operations. The nature of the operation of an object depends on the type of component it represents. That is, objects representing numbers compute arithmetic functions, etc. "At any point in time an object has a state, which is the sum total of its relationships with all other objects in the system." "In systems like Smalltalk the instance variables determine the state of an object and the methods defined in the object's class determine the object's behavior in time." [Ref. 31: p. 6] An object has the following characteristics [Ref. 31: p. 67]:

1. "Objects are temporal, i.e. they exist in time."
2. "Objects are mutable, and have a state."
3. "Objects can be created and destroyed."
4. "Objects are particular, and can be shared."

The first job for the designer programmer is to choose what he she wants to be the objects in the problem he she is trying to solve. Objects can be anything, examples are: numbers, programs, character strings, compilers, computational processes, text editors.

3. Messages

"A message is a request for an object to carry out one of the operations from its own set of operations." [Ref. 30: p. 6] The message specifies which operation it wants, but not how the operation should be carried out. The object to which the message was sent determines how to perform the requested operation. Messages insure the modularity of the system by specifying the type of operation desired, but not how to perform the operation. An object's interface is the set of messages to which the object can respond. Interaction with an object goes through its interface, and its private memory can only be manipulated by its own operations. Messages are the only way to invoke an object's operations. These properties provide security since the implementation of an object cannot depend on the internal details of other objects, only on the message to which it responds. The essential point is that the designer programmer decides on the data structure first, and then which routines are to operate on these data structures. An object's private properties are a set of instance variables that make up its private memory and a set of methods to describe how to carry out its operations.

In the Smalltalk-80 programming system, objects and messages are used to implement the entire programming environment. The designer programmer determines which kind of objects should be described, and which message names provide a useful vocabulary of interaction among these objects. This is an acquired design skill, and it takes time to master. Messages represent the interaction between the components of Smalltalk-80: the arithmetic, control structure, file creation, text manipulation, compilation, and application uses [Ref. 30: p. 40]. The messages make an object's functionality available to other objects, while keeping the object's implementation hidden. The entire programming system becomes accessible as soon as the objects and the messages are understood.

4. Classes

In Smalltalk, you describe a new type of object before creating it. When you are done, the description also works for a whole class of objects. Such an object description is called a class. Any object created from the description is called an instance of the class. A class includes a method for each type of operation its instances can perform [Ref. 30: p. 9].

"The class describes the implementation of a set of objects that all represent the same kind of system component." [Ref. 30: p. 8] In other words a class is just a name for a particular kind of object. The individual class describes the form of its instances, private memory, and how they carry out their operations. The class provides all the information necessary to construct and use objects of a particular kind, including the storage for methods. [Ref. 3: p. 40]

5. Instances

An instance is one of the individual objects described by a class. Each instance has one class, but one class may have multiple instances. Each instance has storage allocated to maintain its own state, and the state is referenced by instance variables. Each object has its own set of instance variables. A class includes a method for each type of operation its instances can perform. In Smalltalk the attributes of an object are represented by instance variables, whose values are themselves objects. All instances of a class represent the same kind of system component. This means [Ref. 30: p. 56]:

1. All instances of a class respond to the same set of methods.
2. All instances of a class have the same number of named instance variables and use the same names to refer to them.
3. An object can have indexed instance variables only if all instances of its class can have indexed instance variables.

6. Methods

A method in a class tells how to carry out the operation requested by a particular type of message. When the special type of message is sent to any instance of the class, the method is executed. The class includes a method for each type of operation its instances can perform. The object's methods can access the object's own instance variable, but not those of any other objects. Methods are simply procedures or subroutines that are invoked by sending messages to a class instance.

A small subset of the methods in Smalltalk-80 are not expressed in the Smalltalk-80 programming language [Ref. 30: p. 9]. These are the primitive methods: they are built into the virtual machine, and cannot be changed by the application programmer. The primitives are invoked with messages exactly like other methods. The purpose of the primitive methods is to allow access to the underlying hardware and virtual machine structure.

C. INFORMATION HIDING

1. Definition

Information hiding is to hide the data structure used in one module from the rest of the program, i.e. prevent access to the data structure from outside of the module. Information hiding is formalized by the following two principles [Ref. 3: p. 292]:

1. "One must provide the intended user with all the information needed to use the module correctly and nothing more."
2. "One must provide the implementor with all the information needed to complete the module and nothing more."

Modularization is the division of a program into a number of independent modules. Each module is like a small program that can be implemented independently of the other modules. The result of each design decision can be hidden in the corresponding module, if this decision is later changed only that module has to be modified. This is called information hiding [Ref. 32].

Modularization can also be viewed as taking a large complex program and splitting it into several little programs using the following principles [Ref. 33]:

1. Cohesion: each module should perform a single complete logical function.
2. Coupling: each module should have access to only those data elements that it needs to complete the assigned task (the "need to know" factor).

This is much like the military methodology of "divide and conquer." Programs can then be designed as a series of linked, single functional modules.

2. Information Hiding in Object Oriented Languages

"In simulation it is often necessary to implement an object in terms of lower level objects. To preserve the integrity of the simulation it is important to distinguish those objects and relationships that are part of the simulation from the objects and relationships that are not part of it." This distinction also facilitates "the modular decomposition of the system." Smalltalk and Simula-67 both allow objects to be used in the implementation of other objects, but they do not enforce the boundary between these levels of abstraction. Enforcement of boundaries have been implemented in more recent object oriented languages, including recent Simula editions. [Ref. 19: p. 12]

Information hiding ensures reliability, testability, and modifiability of a software system by reducing the interdependencies between software components. The internal data structures and procedures can be changed without affecting the

implementation of other modules when the internal state variable of a module is not directly accessed from the outside. Most modern programming languages support information hiding to some degree. The only important exception known to me is ISO Pascal, which provides no way to declare static variables within the scope of a procedure. Standard Lisp does not support information hiding either, but newer implementations of the language do; they use something like the "package" in Ada to support this feature in Lisp. Smalltalk's programming environment uses objects and messages to facilitate modular design. Also other languages use objects and messages for this purpose; Simula uses them for describing simulations, and Hydra uses them for describing operating system facilities in a distributed system.

D. DATA ABSTRACTION

1. Definition

"Avoid requiring something to be stated more than once; factor out the recurring pattern." [Ref. 3: p. 12] People think and understand by means of abstraction, and abstraction is the major technique for understanding, and inventing, of complex structures in the real world. An abstraction provides a simple view of a structure, and summarizes its interesting and important properties. Hoare [Ref. 10: p. 83] stated it like this: "In the development of our understanding of complex phenomena, the most powerful tool available to human intellect is abstraction." Each of the elements of an abstraction itself can be an abstraction for details at a lower (or higher) level. At a given level of abstraction the view must be simple or the abstraction is inappropriate. The major idea is to use abstractions to make it easier for the human being to understand a complex problem. Data abstraction is the ability to define a type by specifying the operations that are meaningful for it, without exposing the representation of the type.

In general the introduction of an abstraction layer will reduce the efficiency of the algorithms. Optimized compilers can remove this problem, but in dynamic binding languages, like Lisp and Smalltalk, the added functionality means a significant loss in performance.

2. Data Abstraction in Object Oriented Languages

According to our definition of object oriented programming languages Pascal and Ada are not object oriented because they do not support all four features in the definition. Data abstraction in object oriented programming languages are discussed in relation to Pascal and Ada in this section.

Some languages, like Pascal, allow the programmers to define new data types, but the representation (e.g. array, record etc.) of the type is not hidden from other parts of the program. When the representation is exposed, the abstract properties cannot be assured. Given an object of a program defined type, other parts of the program can access the representation directly and hence can violate the abstraction properties. Object oriented programming is also an abstraction mechanism. In some object oriented programming languages, like Smalltalk, objects that have a lot in common are grouped together in one class. This is an abstraction, but also it shows how data abstraction can be considered a way of using information hiding. Some people [Ref. 22: p. 11-19] view the important features of object oriented languages as information hiding and data abstraction. This definition would include Ada in object oriented languages. Our definition of object oriented programming languages excludes Ada, but because it is one of the newer procedural programming languages it is studied. Ada includes abstractions for both control and data. The control abstraction includes procedures and functions, but also assignment statements, if statements, block statements, etc. [Ref. 34: p. 9]. Ada uses several features, operator overloading, generic program units, and packages, to implement data abstraction. The data abstraction in Ada is established at compile time.

In, for example, Smalltalk the internal structure can be hidden from other objects. This data abstraction can be illustrated with the following example: Suppose a program is written with the purpose of playing chess, or as in this case restricted to movement of the chess pieces. A Smalltalk program would invoke a method "move_to," passing the destination square as a parameter. The point is that an assignment statement is not used to modify the data structure describing the chess pieces' positions on the board. The main advantage is that both the representation of the chess pieces and the implementation of "move_to" can be changed without altering the code in other objects that access them.

Dynamic binding seems to be the only reasonable solution to the data abstraction problem if a more general reuse of code is wanted. It is possible to write

general procedures that use variables and procedures supplied by the caller's environment.

E. DYNAMIC BINDING

1. Definition

"With dynamic binding the meaning of statements and expressions are determined by the dynamic structure of the computation evolving in time, that is at run time. With static binding the meaning of statements and expressions are determined by the static structure of the program." [Ref. 3: p. 119] In static scoping a procedure is called in the environment of its definition, but in dynamic scoping a procedure is called in the environment of its caller. Scoping rules apply uniformly to all names, not only for variable names. One of the advantages of dynamic scoping is that it is possible to write a general procedure that makes use of variables and procedures supplied by the caller's environment.

2. Dynamic Binding in Object Oriented Languages

The object oriented programming style pushes the responsibility for interpretation of the message onto the objects themselves. Smalltalk-80 has dynamic binding of methods to a message based on the class of its receiver. This dynamic binding requires a lookup of the selector in the message dictionaries of the superclass chain for the receiver. Each object is sent exactly the same message selector, but the object itself determines how to perform the requested operation. This ensures that implementation of an object cannot depend on the internal details of other objects, only on the message, to which they responds. The same message can elicit a different response depending on the receiving object. Smalltalk-80, for example, is a dynamically typed programming language, and is therefore generally harder to compile than interpret. Operator overloading in Ada does not have this form of dynamic polymorphism since the address of the procedure invoked is fixed at compile time [Ref. 35: p. 142].

F. INHERITANCE

1. Definition

Inheritance is the sum total of "genetic characteristics derived or acquired from ancestors." [Ref. 36] Inheritance is especially concerned with the management of change, and it is the key to simulation, and reuse of code.

2. Inheritance in Object Oriented Languages

Classes are related to one another by an inheritance relationship, and inheritance is fundamental to the object oriented paradigm. In Smalltalk, for instance, inheritance is interpreted as follows [Ref. 37: p. 89]:

1. If class B inherits from class A, then objects of class B supports all operations supported by objects of class A.
2. If class B inherits from class A, then class B's instance variables are a superset of class A's instance variables.
3. If class B inherits from class A, then the code of any methods not explicitly written for class B will be obtained from class A.

Inheritance can be used to define a class in terms of one or more other classes. If a class B inherits directly from a class A, we say that A is the parent of B and that B is the child of A. The terms ancestor and descendant are used as normal, and follow the inheritance chain.

See Figure 5.1 for a visual representation of inheritance. In this case class B inherits from class A, and B is the child of parent A. Inheritance enables programmers to create new classes of objects by specifying the difference between a new class and an existing class. Object types can have ancestor object types from which they inherit characteristics, and the descendant type can change characteristics inherited from its ancestor. Some object oriented systems allow inheritance between all the objects in the system, but normally only inheritance between classes are allowed. A class may be modified to create another class. The class that creates the other class is called the superclass, and the other class is called the subclass. The subclass inherits everything about its superclass. The terms subclass and superclass are often used ambiguously to mean both direct and indirect inheritance. In CommonObjects [Ref. 38], for instance, the distinction between direct and indirect inheritance is particularly important because inheritance is a mechanism for defining objects whose interfaces include all the operations defined for another class without saying anything about the internal representation. In Smalltalk, for instance, inheritance is primarily a mechanism for

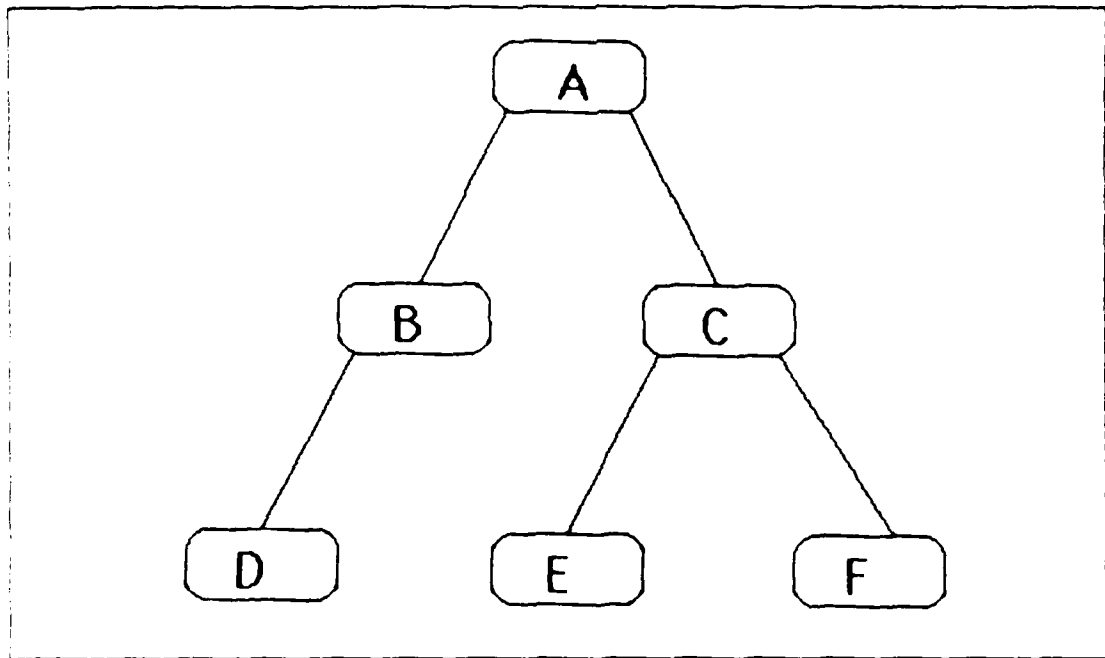


Figure 5.1 Inheritance Example.

building more complex code bodies out of simpler ones, and more complex data structures out of simpler ones. A large amount of code can therefore in Smalltalk be reused because it is not necessary to start from scratch each time.

G. SOME ADVANTAGES AND DISADVANTAGES IN OBJECT ORIENTED PROGRAMMING

1. Advantages

In general the object oriented programming process is closer to the high level application programming than most traditional techniques. The main point seems to be that it is easier to start with an idea, and from the idea design code more easily follows because we do not need a complete specification of the problem to get started. Existing objects from other programs or libraries can be used to form new programs instead of always starting from scratch. This generally saves time. Normally the objects we start with are bug free, the specialization, changes we add to them will therefore be easier to perform and the time spent on debugging decreases. Information

hiding is supported so each of the objects can be viewed as a closed universe, therefore the design code is more orderly. And finally the programs are easier to maintain.

In general we do not have to worry so much about the details of the algorithms, but rather work with the description of the properties of the objects we use to build the program. We are at a higher level of abstraction and specifications of behavior, i.e. we worry about what to do instead of how to do it. The hardware resources are taking over much of the work previously done by the designer programmer.

2. Disadvantages

MacLennan [Ref. 19: p. 3] states the following general disadvantages of object oriented programming:

1. "It is difficult to reason about things that change in time."
2. "Object oriented languages provide little ability for algebraic manipulation."
3. "The analysis of object oriented programs can be hard."

In addition it is hard to master object oriented languages, like Smalltalk, because there is such an enormous number of objects available in the system. It is generally easy to get started using a subset of the system, but to use all the built in capabilities in the library takes a long time to learn and master. The naming convention in Smalltalk is generally better than in Lisp, i.e. the names used on objects gives a good indication of the purpose of the objects. Lisp also has a large number of functions, often with names that do not give a good indication of the purpose of the functions.

The Smalltalk-80 system is not designed to run background processes, or to be run on a time shared computer system. Much of the reason for this is all the loops in the source code waiting for a pointing device (mouse) input, i.e. the cost of a user friendly window interface.

H. SUMMARY OF THE CHAPTER

The main paradigm of object oriented programming is simulation. Object oriented languages are organized around objects, and group data structures and routines together into objects. Smalltalk-80 is a typical example of object oriented programming languages that supports the four criteria in this thesis: information hiding, data abstraction, dynamic binding, and inheritance hierarchy.

There are five major concepts in Smalltalk: object, message, class, instance and method. System components are represented by objects, objects are instances of classes, objects interact by sending messages, and messages causes methods to be executed.

The main advantages are that we can program at a higher level of abstraction and specification of behavior, in addition the dynamic binding makes it possible to write general procedures that use variables and procedures supplied by the caller's environment. The disadvantages are long learning time due to large numbers of objects and classes. The dynamic binding makes the system slow.

VI. INHERITANCE

A. BACKGROUND

"In a simulation of any complexity it is infeasible to describe the behavior of every individual object," because the simulated "world" consists of such an enormous number of objects [Ref. 31]. Due to this infinite mass of information it is impossible to represent all the details about the whole universe inside a computer system. It is easier to group objects into classes of similarly behaving individuals (abstraction), so that their common behavior can be described just once. "Abstraction is the decision to concentrate on properties which are shared by many objects or situations in the real world, and to ignore the differences between them." [Ref. 10: p. 84] In object oriented programming languages, like Smalltalk, all computation is viewed as simulation, the paradigm supported in this thesis, and programming language objects correspond to real world objects. The purpose of a computer program is normally the modeling of some aspect of the real world, often involving the changing relationships among real world objects. Abstraction for the simulation, and selection of relevant subparts of the universe, is needed in order to design and code a computer program. "The state of the simulation is represented by a finite number of objects connected by a finite number of relationships." [Ref. 3: p. 9] When a program executes, objects may be created or destroyed, and the relationships among them may change. In addition to the real world objects we have the "nonreal" objects that can be exemplified by the "what if" questions in spreadsheet applications.

B. INTENSION VERSUS EXTENSION

Because the computer is not able to represent the whole universe in a simulation it is necessary to distinguish between intension and extension. In an object oriented language, like Smalltalk, the entire state of the simulation is therefore represented by a finite number of objects connected by a finite number of relationships. "Two relations

have the same intension if they are supposed to model the exact same external relationships or properties. They have the same extension if they apply to the same object." [Ref. 31: p. 12] and extensions may vary over time. Two relations may be extensionally the same even though they are intensionally different. The intension of a computer object is the real world object it is modeling; the extension of a computer object is the set of relations to which it belongs. Objects with different intensions may coincidentally have the same extension. Therefore two or more computer objects that are intended to model distinct real world objects may agree in all the modeled properties and relationships. This may happen because a computer system is unable to represent every property and relationship in the universe. The designer programmer must select a finite number of these objects and relations that are relevant to the problem he/she wants to solve. The programming language system may have problems distinguishing intensionally distinct relations that happen to have same extension.

C. INHERITANCE IN GENERAL

1. Inheritance versus Data Abstraction

Object oriented programming encourages modular design and software reuse. Data abstraction is the ability to define new types of objects whose behavior is defined abstractly. Normally an object oriented language supports data abstraction by preventing an object from being manipulated by other means than via its defined external operations. The fundamental idea of inheritance is that software modules may be defined as extensions (specializations) of previously defined software modules. The original software module does not have to be modified when it is used as a basis for a new extension. In object oriented programming where the basic software modules are based upon abstractions, an extension of a software module would then correspond to a refinement (specialization) of hierarchies of abstractions.

Inheritance compromises encapsulation (modularization) in many object oriented languages. For example Smalltalk lets the programmer access the inherited instance variables. The benefits from encapsulation are improved understandability of programs and easier program modification. To be able to do debugging and creation of programming environments most programming languages provide ways to circumvent encapsulation. In Smalltalk, for example, the operation "instVarAt" and

"InstVarAtInput" allow access to any named instance variable of any object [Ref. 30: p. 247]. This is special features that will not be covered more in detail. Inheritance complicates the situation by introducing a new category of client (user) for a class. In addition to clients that simply instantiate objects of the class and perform operations on them, also other clients (class definitions) inherit from the class. Venn diagrams will be used as a descriptive tool in this section in order to clarify the inheritance problem. [Ref. 39: p. 38].

Many of the ideas in this section of the thesis builds on work done by Adele Goldberg and David Robson.

2. Subclassing

Single inheritance is the case where the inheriting class (the child), directly inherits from a single class, the parent. Hierarchical classification is to factor out common behavior of several classes of objects. Smalltalk and Simula do this by permitting classes to be subclasses of other classes. In our definition of object oriented programming the feature inheritance hierarchy is the one that supports the simulation paradigm, and makes the object oriented languages so special. Without inheritance the addition of a new type of object requires writing entirely new procedures for common operations. There will be a great deal of similarity between these different methods, but there will be a need for continuous rewriting of methods that differ slightly or not at all. Inheritance can reduce this burden, and drastically reduce the number of lines of code in a program. Smalltalk-80's class hierarchy builds on run time checking, and run time binding of messages to methods.

David Sandberg [Ref. 40] describes an alternative to subclassing that uses compile time typing, adds parameters to classes, and introduces a new form of class called a descriptive class. This alternative supports building larger modules from smaller modules, while Smalltalk-80 encourages refining the behavior of an existing class by creating subclasses. Like subclassing, the descriptive classes allow sharing of code. The subject of descriptive classes will not be pursued further in this thesis.

The class structure described so far does not explicitly provide for any intersection in class membership, see Figure 6.1 for a visual representation. None of the classes in the figure overlap, so there are no shared objects between classes. In the Venn diagrams used in this section, the circles represent the classes and the black dots the instances. The representation of an instance of a class would then be a black dot within a circle.

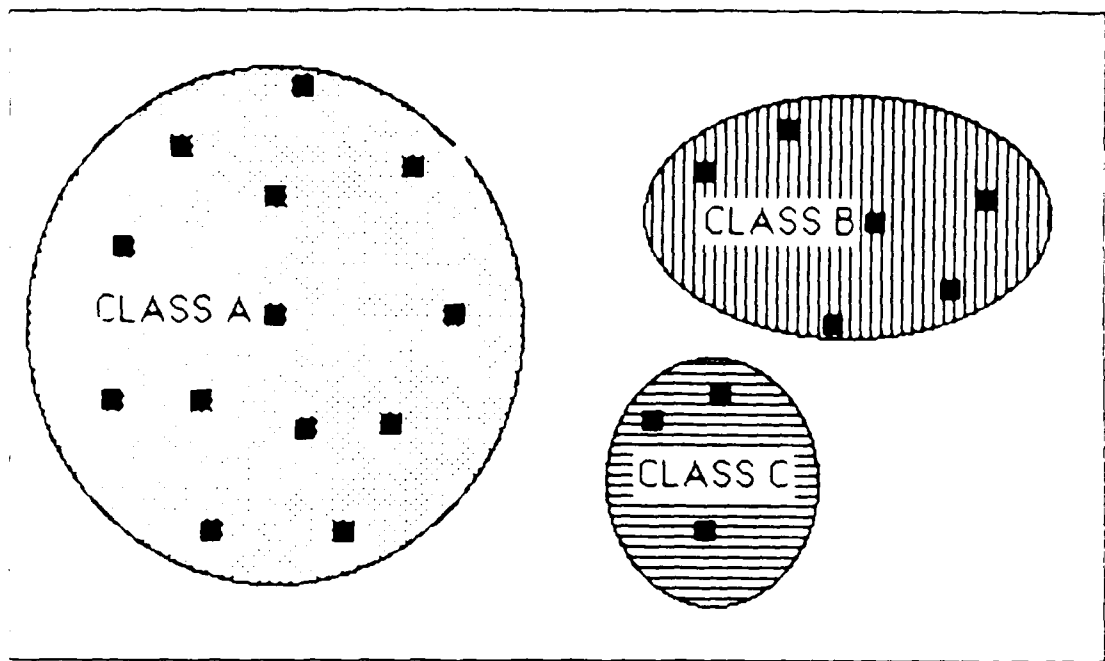


Figure 6.1 Without Intersection in Class Membership.

The task of creating a specialization of an existing class is called subclassing, and the existing class is a superclass of the new class. The classes in Smalltalk-80 form a tree; more than one class may share the same superclass, but each class has only one immediate superclass. The root of the tree is the class "object," this is the only class without a defined superclass. In general the instances of a superclass cannot be affected by the existence of one or more subclasses. Therefore variable names added to a subclass must be different from any variable declared in the superclass. The subclass inherits instance variables, class variables, and methods from its superclass. The subclass may add instance variables and class variables to make the subclass more specialized than the superclass from which it derives. A subclass can also override or provide additional behavior to methods of a superclass. Methods are overridden when a new method for an old method's selector is provided [Ref. 35: p. 142]. If a new method makes use of the old method in Smalltalk a message-send to the pseudovisible "super" is embedded in the new definition of a method. The new behavior may precede, follow, or surround the existing behavior.

If, in Smalltalk, during a message-send a method is not found in the immediate class of the instance, the superclass method dictionary is checked, if not

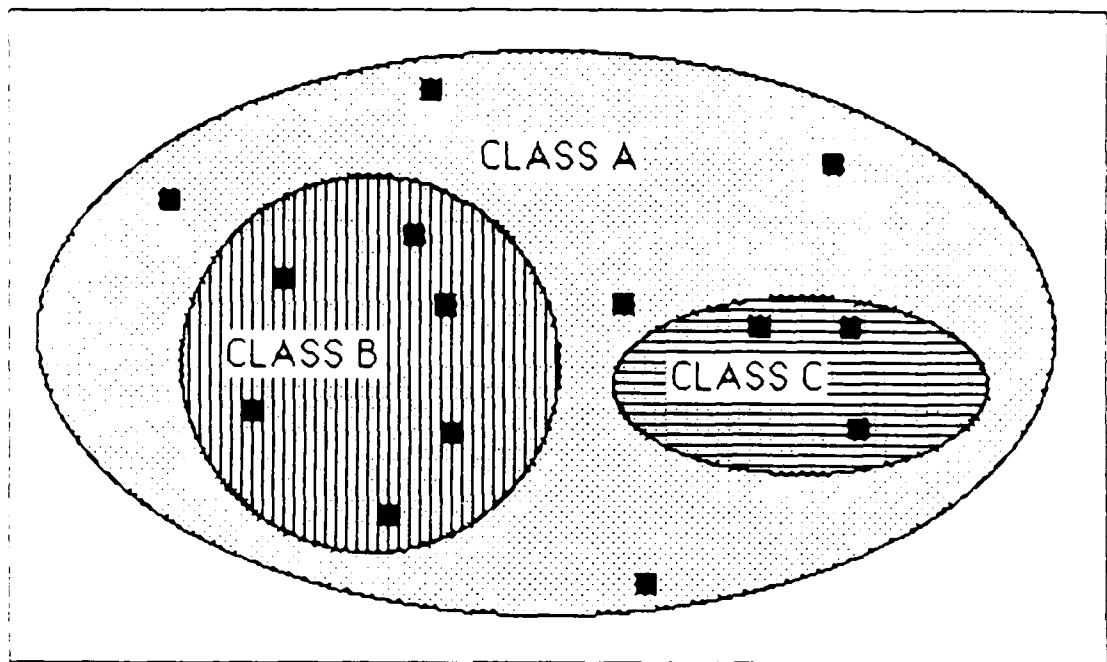


Figure 6.2 Subclasses.

found there either the superclass's superclass is searched and so on until the "object" (the root of the tree) is searched. If even the class "object" does not have a matching selector then an error message is returned.

Subclassing is to allow a class to include all instances of another class, but not to allow more general sharing in class membership (i.e. not multiple inheritance), see Figure 6.2 for a visual representation. A subclass specifies that its instances will be the same as instances of another class, its superclass, except for the differences that are explicitly stated. Smalltalk-80's subclassing is a pure hierarchical system; i.e. if there are any instances of a class that also are instances of another class, then absolutely all the instances of that class must be instances of the other class. In Smalltalk the classes themselves are considered to be objects belonging to a meta class. A meta class is a class whose instances themselves are classes. There is a one-to-one correspondence between a class and its meta class.

The use of classes and metaclasses provides a mechanism for sharing information between different objects via inheritance. Inheritance is not the only scheme for information sharing, and the requirement that each object permanently belongs to a class imposes constraints on the mutability of the behavior of an object.

An example of another mechanism for information sharing is "delegation" [Ref. 41: p. 60]. Using Delegation, subcomputations can be passed on by an actor to another actor which continues the processing. Delegation provides a mechanism for code sharing where the control is passed to an independent actor. In inheritance mechanisms, on the other hand, information may be requested from a more general class to which an actor belongs, but the control remains localized. Actors are computational agents which carry out their actions in response to incoming messages. Actors encapsulate procedural and declarative information into a single entity. High level actor languages use inheritance for conceptual organization and delegation for structuring the sharing of code between different actors.

CommonObjects [Ref. 39] is an extension of Common Lisp, and it is representative of a new generation of object oriented programming languages that build on the Smalltalk and Zetalisp experience. CommonObjects provides strong support for encapsulation, in particular with respect to inheritance. Classes are not objects in CommonObjects, just as types are not objects in Common Lisp. The access of a subtype to its supertypes is restricted to the same abstract interface as that presented to users. Multiple inheritance, for example, from two classes with identical named instance variables result in separate copies of such variables when an abstract user interface is used, but only a single copy if a non abstract user interface is used. Inheritance, in CommonObjects, is a mechanism for defining objects whose interface include all the operations defined for another class, without saying anything about the internal representation.

3. Inherited Instance Variable

Generally in object oriented languages the code of a class can directly access all the instance variables of its objects. This is true also for the instance variables defined by an ancestor class. Permitting access to instance variables defined by ancestor classes compromises the encapsulation, and therefore weakens one of the major benefits of object oriented programming. This problem with instance variables has resulted in many different implementations [Ref. 39: p. 40]. Flavors [Ref. 27] does not allow merging of inherited instance variables and instance variables defined locally in a class. Smalltalk signals an error if a class defines an instance variable with the same name as an inherited instance variable. The subclass (in Smalltalk) inherits both the variable declarations and methods from its superclass, but gives itself a new class name. The existence of a subclass cannot affect the superclass, therefore it is illegal to add a

variable name to a subclass that is declared with the same variable name in its superclass. Addition of shared variables will make them accessible to the instances of the subclasses of the class. This also means that a subclass has the same, or a larger number of variables than the superclass (i.e. the subclass is more specialized).

4. Programmer's View of Inheritance

It is not obvious that the designer of a programming language, or a programming environment, thinks of inheritance the same way a user does. For the designer programmer the purpose of inheritance can be the following [Ref. 39: p. 41]:

1. A private decision taken by the designer programmer to reuse code because it is useful (saves time) to do so for him/her. At the same time it should be easy to change such a decision later on.
2. Making a public declaration that objects of the child class obey the semantics of the parent class. This means the child class is a specialization of the parent class. Brachman covers this in depth in the context of knowledge representation [Ref. 42: p. 80-93].

For the programmers in object oriented programming, a single object may look different in different cases, i.e. multiple views. That is because the programmers use different parts of the object, and manipulate it differently. When the multiple views in addition are used in an inheritance hierarchy, there are some problems with how the programmer understands his programming environment, which consists of a large number of objects.

In an integrated interactive programming environment it is important that the user interface is consistent, and as powerful as possible. Multiple inheritance, without restrictions caused by the computer system, is the most natural to use for the programmer. On the other hand it is very difficult to implement multiple inheritance in an integrated software system.

D. MULTIPLE INHERITANCE

1. Overview

The most general way to achieve multiple inheritance in object oriented languages is to allow arbitrary intersection of class boundaries, like in Figure 6.3 which shows arbitrary intersection of classes. Multiple inheritance allows a situation in which some objects are instances of two or more classes, while other objects are instances of

only one class or the other. Multiple inheritance is a very powerful technique for reusability of code, allowing the combination of more than one previously defined class. Multiple inheritance presents problems in terms of what is to happen if there are multiple paths or conflicting instance variables. How can we make changes to the inheritance hierarchy safely? If the use of inheritance itself is globally visible (as in most implementations) changes to the inheritance hierarchy cannot be done safely.

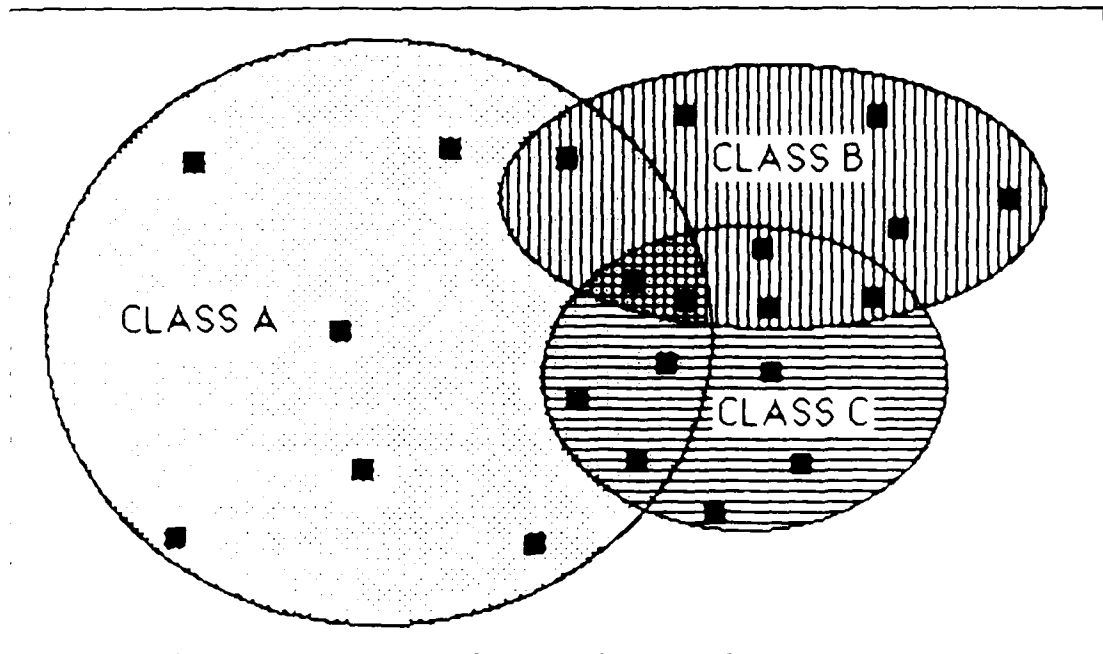


Figure 6.3 Multiple Inheritance.

In Smalltalk-80 the implementation description can be modified by a subclass as follows [Ref. 30: p. 59]:

1. The class name must be overridden.
2. Variables may be added.
3. Methods may be added or overridden.

To override a method means that if a subclass adds a method with the same selector as a method in the superclass, the subclass's instances will respond to messages with that selector by executing the new method. Standard Smalltalk does not support general multiple inheritance, it uses a tree structure, i.e. each class has only one superclass. The pure tree structure used in Smalltalk, and the fact that there are no hidden side

effects upon other objects makes the language easy to use. In the tree structure, essential information is highlighted on one level and the details are specified on a lower level. On all the layers of the tree but the lowest, the objects are fairly complex. Many of the ideas in this section is from Alan Snyder's work, especially from his paper: "Encapsulation and Inheritance in Object-Oriented Programming Languages" [Ref. 39: p. 38-45].

Multiple inheritance means the class can have one or more parents (superclasses). A class can be viewed as forming the root of a directed acyclic inheritance graph, where each class is a node and there is an arc from each class to its parent. "In Smalltalk, programming language objects are grouped into classes (i.e., abstractions) of similarly behaving objects." [Ref. 31: p. 5] An example of multiple inheritance is shown in Figure 6.4 which use an acyclic graph as an illustration of the problem.

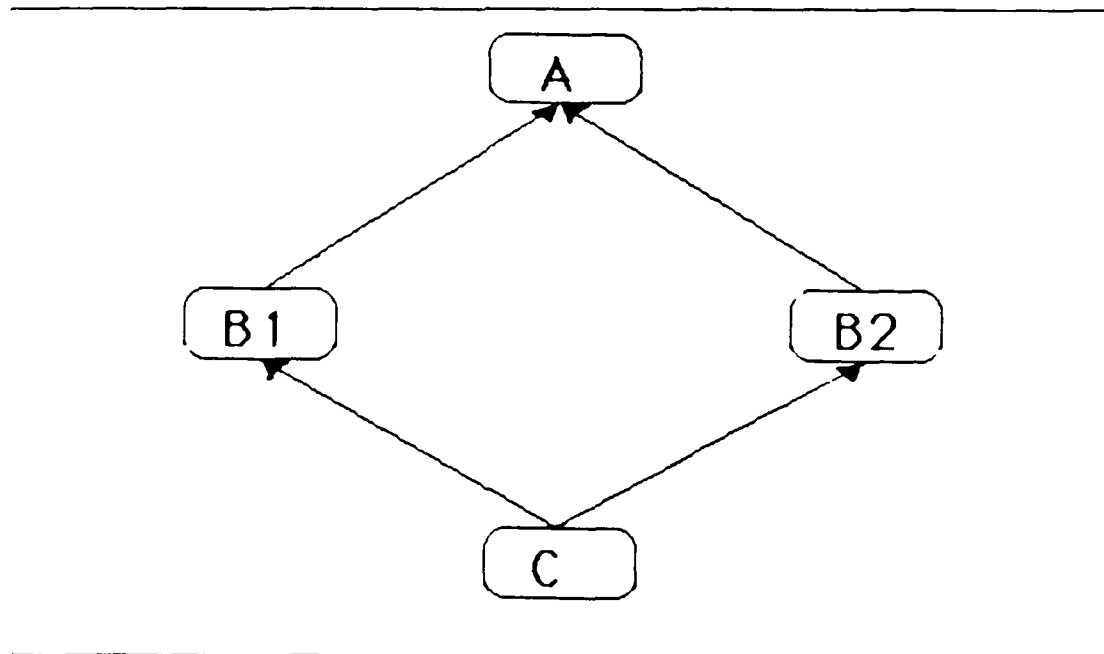


Figure 6.4 Example of Multiple Inheritance Acyclic Graph.

There are three strategies in common use that try to solve the problem with multiple inheritance. The first tries to deal directly with the acyclic inheritance graph. The second first flattens the graph into a linear chain, and then uses the rules for single

inheritance. The third, the tree solution, avoids the problems of graph oriented and linear solutions, by duplicating nodes. These three solutions will next be covered in more detail.

2. Graph Oriented Multiple Inheritance Solution

Examples of object oriented languages that model the inheritance graph directly are; extended Smalltalk and Trellis Owl. The operations in these two languages are inherited along the inheritance graph until redefined in a class. The inheritance problem arise when a single class is reachable from another by multiple paths as in Figure 6.4 where the graph is not a tree, but an acyclic graph. [Ref. 39: p. 42]

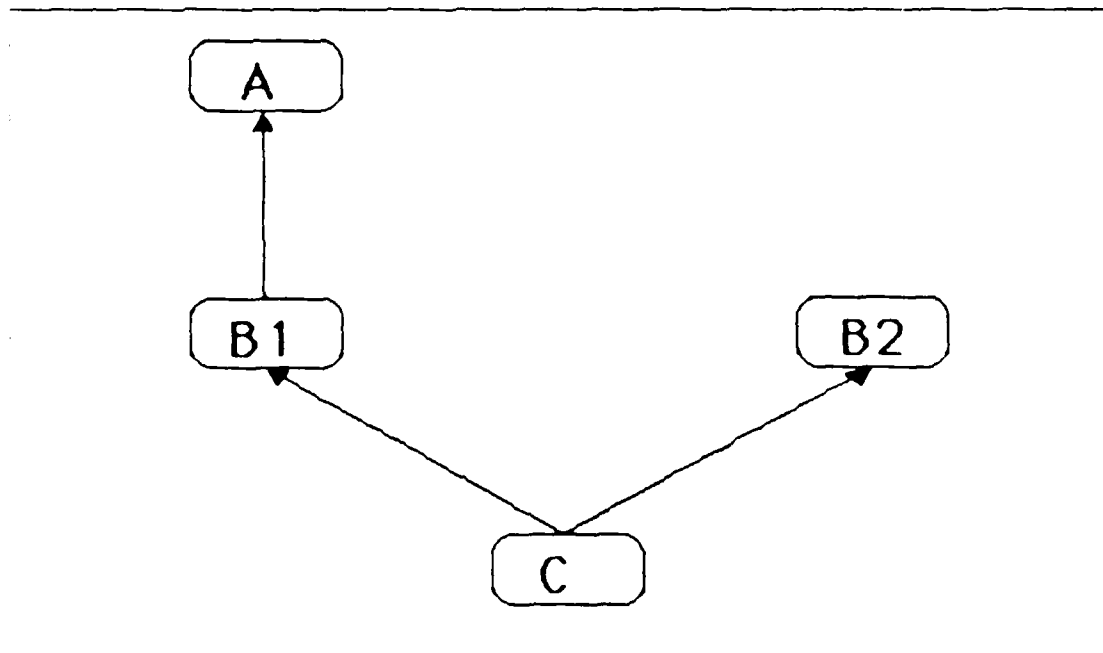


Figure 6.5 Example of Altered Graph Oriented Multiple Inheritance.

In multiple inheritance one class can have more than one parent. If there is more than one parent, and the class inherits operations with the same identity (name) from more than one parent, there is an identity conflict. This identity problem can be solved, see Figure 6.5, by redefining of the operations in the subclass (child). Disadvantages with the graph oriented solution are that for any ancestor class there are defined only one set of instance variables, regardless of how many paths there are to reach the class in the inheritance graph. This limits the programmer's freedom to use

the inheritance within a class without the possibility of destroying some descendant class. The problem is most serious if the operation is invoked on the same set of instance variables more than once, and if the operation on that specific class has side effects.

3. Linear Chain Multiple Inheritance Solution

Flavors and CommonLoops are examples of languages that use linear solution to solve the multiple inheritance problem when the graph is not a tree, but an acyclic graph. These two programming languages "first flatten the acyclic graph to a linear chain, without duplicates." Thereafter the result is treated as single inheritance. See Figure 6.6 for a visual representation of the solution. Algorithms in the language create a total ordering that preserves the ordering along each path through the inheritance graph, but unrelated classes may be inserted between a class and its original parent. The computed inheritance chain may have the property that parent of a class B may be a class A with unknown content to the designer of class A. [Ref. 39: p. 43]

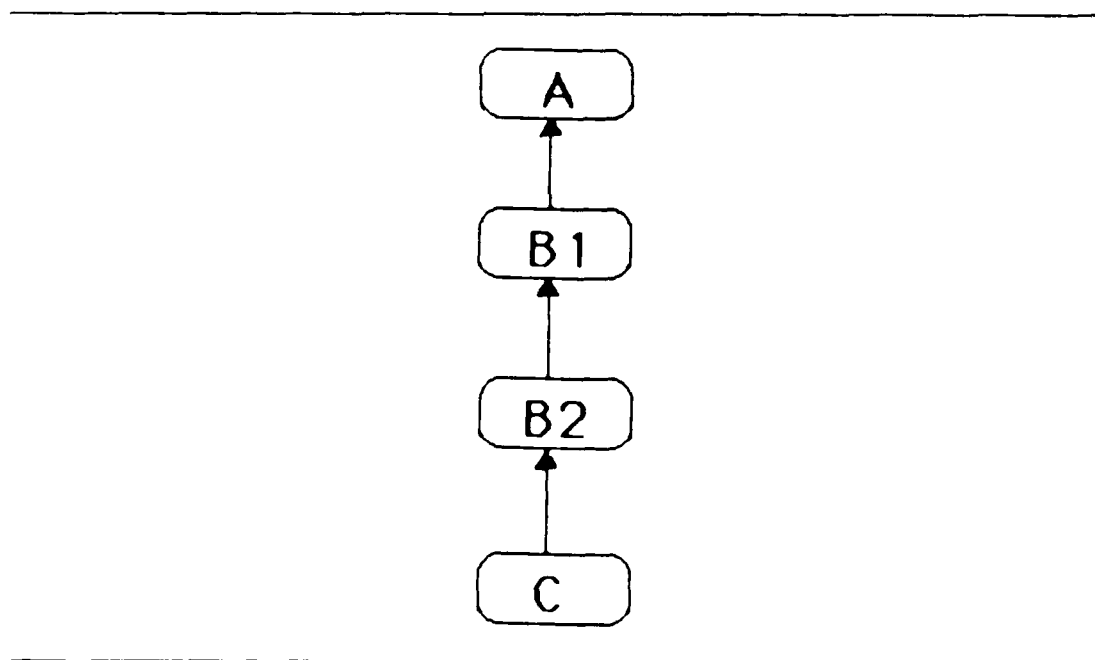


Figure 6.6 Example of Linearized Chain Multiple Inheritance.

One disadvantage with the linear solution is that: if more than one parent defines the same operation, one operation will be selected. Other problems arise from

the fact that unrelated classes may be inserted between a class and its original parent. This inserted class may redefine some operation so that the communication between the child and the original parent is broken.

4. Tree Conversion Multiple Inheritance Solution

Both the graph oriented and the linear solutions have some drawbacks, as described earlier. The tree solution avoids these problems by duplicating nodes, see Figure 6.4 for a visual representation of the solution. There are still unresolved problems with multiple inheritance because the inheritance is not close enough to the user's view of multiple inheritance.

CommonObjects is an example of object oriented languages where the semantics models the inheritance graph [Ref. 39: p. 44]:

1. Regardless of the source it is illegal to inherit an operation from more than one parent.
2. Each parent of each class defines a completely separate set of inherited instance variables. The acyclic graph is converted into a tree by duplicating nodes, and each path creates a separate set of instance variables.

Situations are therefore avoided where an operation can accidentally be invoked multiple times on the same set of instance variables, or where two classes conflict in their use of an inherited class. In CommonObjects these problems are corrected by restricting the access to the inherited instance variables. The access is provided in the form of operations, and the multiple inheritance supports encapsulated class definitions.

E. SUMMARY OF THE CHAPTER

Inheritance in object oriented programming languages facilitates reuse of code. Software modules may be defined as extensions (specializations) of previously defined software modules.

Single inheritance is the case where the inheriting class directly inherits from a single class, the parent. In our definition of object oriented programming the feature inheritance hierarchy is the one that supports the simulation paradigm, and that makes the object oriented languages so special.

Subclassing is to allow a class to include all instances of another class, but not to allow more general sharing in class membership, i.e. not multiple inheritance. The

more general way to achieve multiple inheritance in object oriented languages is to allow arbitrary intersection of class boundaries. There are three commonly used methods to solve the problem of implementing multiple inheritance: graph oriented solution, linear chain solution, and tree conversion solution.

VII. INTERACTIVE PROGRAMMING ENVIRONMENT

A. WHAT IS AN INTERACTIVE PROGRAMMING ENVIRONMENT?

1. Definitions

An interactive programming environment is a software system that manipulates software programs while "cooperating" with the user. A software system is a set of software tools that can aid the user in his/her job. These tools include [Ref: 43]

1. Programming language:
 - a. interpreter
 - b. compiler
 - c. code generator
 - d. type checker
2. Runtime debugger
3. Emulator
4. Specification tools
5. Documentation tools:
 - a. word processing
 - b. text editing
 - c. spelling checker
 - d. graphics tools
6. Preparation tools:
 - a. syntax directed editor
 - b. program formatter (pretty printer)
 - c. application generators (4th generation languages)
 - d. conversion tool (between languages)
 - e. cross reference generators
 - f. automatic flow chart, hierarchy diagram etc.
 - g. screen management tools
 - h. library and library management
 - i. source code management (linkers, loaders and locators)

7. Maintenance tools.
 - a. test program generators.
 - b. tracers.
 - c. dumps and dump interpreters.
 - d. version control.
 - e. source code control.
8. Performance tools.
 - a. measurement tools (histogram generators).
 - b. analytic performance estimator.
 - c. benchmarking.
 - d. optimizers (both object and source code).
9. Minor tools.
 - a. copy utilities.
 - b. directories.
 - c. file differencer.

2. Impact of Tools

Software tools are becoming more and more important in developing new application areas for computer systems. This transformation to interactive integrated programming environments provides the user with conversational access to data. Integration means that the different tools are designed to "fit together," and are able to use each other's features. The integration increases the power of the environment because the sum total of the system is larger than the sum of the single tools. The increased group of users often view the system only as a tool that may help them in their job. James Burke stated it once [Ref. 44]:

But the moment man first picked up a stone or a branch to use as a tool, he altered irrevocably the balance between him and his environment. From this point on, the way in which the world around him changed was different. It was no longer regular or predictable. New objects appeared that were not recognizable as a mutation of something that had existed before and as each one emerged it altered the environment not for a season, but forever. While the number of these tools remained small, their effect took a long time to spread and to cause change. But as they increased, so did their effects, the more tools, the faster the rate of change.

The designer's/programmer's environment influences, and often determines, what kind of problems he/she can and will want to solve, how far he/she can go, and how fast. Therefore the programmer can devote more time and energy to the task of programming itself, i.e. to conceptualize, design and implement a program, and he

more ambitious and more productive. The available programming tools are a critical part of the environment, and much work today is going on to implement and test new tools.

3. What is so Special about Programming Environments

Historically programming environments have been used to help the designer-programmer in the development and maintenance of software. In the beginning they used noncomputerized tools like coding forms, pencils etc. Soon this evolved into computerized tools like assemblers, compilers and high level programming languages. The designers and programmers built tools to help themselves: symbolic debuggers to aid in the debugging process, operation systems and file systems to manage the computer system, text editors to ease entry and modification of the program text, and so on.

The unique thing about an interactive integrated programming environment is that it is used to represent and develop other programs, and to manipulate these other programs. Many of the abstract 'types' are entities in the program the programmer is developing in the interactive programming environment. The distinction between the program that is manipulated, and the programming environment in which it is being manipulated becomes less and less over time. In an interactive integrated programming environment the programmer may interweave activities without losing accumulated information, and without giving up capabilities. Typically, in programming environments, computer resources are doing the work in order to save human resources.

Today we have a variety of programming environments, but there exist no well defined classification of them. Burstow and Shrobe [Ref 4, p. 500] suggest the following three classes:

1. Programming environments concerned with the entire lifecycle of a program or system.
2. Programming environments concerned primarily with the coding phase and whose tools are relatively independent of each other.
3. Programming environments which regard coding, debugging, testing and maintenance as a single process of program development through incremental enrichment.

The discussion in this chapter concentrates on the third class which seems to be closer to what we want in the near future. We are in this context the designers-programmers and not the managers. The third class covers the interactive

integrated programming environments like Interlisp and Smalltalk. These two environments are relatively easy to extend to take care of the whole life cycle also (i.e. class one).

B. IDENTITY OF OBJECTS

1. Definition of Identity

Identity is that property of an object which distinguishes each object from all others. [Ref. 45: p. 406] Two dimensions, at least, are involved in identity, the representation dimension and the temporal dimension. The representation dimension classifies the programming languages based on whether they represent the identity of an object by "a user specified name, by its value, or if it is built into the programming language itself." [Ref. 45: p. 414] The temporal dimension classifies the programming languages based on whether they preserve their representation of identity "within a single program or transaction, between transactions, or between structural reorganizations." [Ref. 45: p. 407]

2. Identity in Interactive Programming Environments

Most programming languages do not differentiate between addressability and identity, and use the variable names as the only way to distinguish temporary classes (objects).

Smalltalk-80 implements identity with the "loop" (object oriented pointers). An "loop" is an entry in an object table, and we say that the identity is implemented through a level of indirection. Indirection, i.e. indirect physical or virtual address implementations would allow individual classes (objects) to be moved within one address space. This would provide full data independence, but not allow sharing of classes (objects) among multiple programs. The "loops" are the most common form of data manipulated by the Smalltalk-80 interpreter.

In object oriented systems, like Smalltalk, the instance variables determine the current state of an object while the methods defined in the object's class determine how the object behaves. It is possible that two or more objects may be identical in all properties in a simulation, while representing different real world objects. The identity is independent of the object's state, and therefore distinguishes each object even in the case where they may be temporarily or permanently identical in all properties with one other object.

The identity is very important when building interactive integrated environments using object oriented programming languages. The tools are using each other, therefore distinct identities are critical to prevent identity conflicts.

3. What Language to use in an Interactive Programming Environment

In theory all programming languages are equally powerful. To actually write the code in different languages may be more or less easy to do for the programmer, but that is irrelevant as a theoretical measure of power.

In general, programming languages are more suitable for certain jobs than others: Basic is easy to learn and is good for small dialogue oriented applications. Fortran is well suited for numerical applications. Cobol is tailored to business data processing. Pascal is designed for teaching structured programming. Ada is ideal for large embedded systems. Lisp is very good for processing symbolic information. Smalltalk is designed for simulation. APL for manipulation of vectors and matrices. C for system programming. Simula for discrete simulation, etc.

An interactive integrated programming environment consists of a set of computerized tools that is designed to help the user of the system. The programming environments take cognizance not only of the technical nature of the software construction process, but also of the social environment in which it is actually used. The programming task takes place in various managerial and social settings, so the computerized tools appropriate in one context and may be inappropriate in another.

Currently it seems to be easier to implement an interactive integrated programming environment in an interpreted language with dynamic binding than in a traditional compiled language. Lisp and Smalltalk will therefore be covered in more detail in relation to interactive integrated programming environments. In Smalltalk-80 the incremental program execution of Lisp is combined with Simula's class subclass and virtual concepts. Smalltalk and Lisp have a lot in common: a flat set of definitions (classes), dynamic name binding and run time type checking. Both Lisp and Smalltalk have a main loop written in itself, Smalltalk or Lisp respectively. The loop: read a command, execute the command, print the result, and loop. And both languages support exploratory software development.

4. Incremental Program Development

Complex problems are often difficult to specify and design. We often know we have a problem, but exactly what causes the problem and how to solve it may well be more difficult to find. A software program will normally go through a series of changes over its life cycle. In the beginning it may exist as a loose mental description of what the designer programmer wants the program to perform. This in turn may (or may not) evolve into a more formal specification, which in turn may evolve into a design, and finally become code in some programming language. Maintenance may also be done in the same fashion. See Figure 7.1 for a visual presentation of this incremental program development.

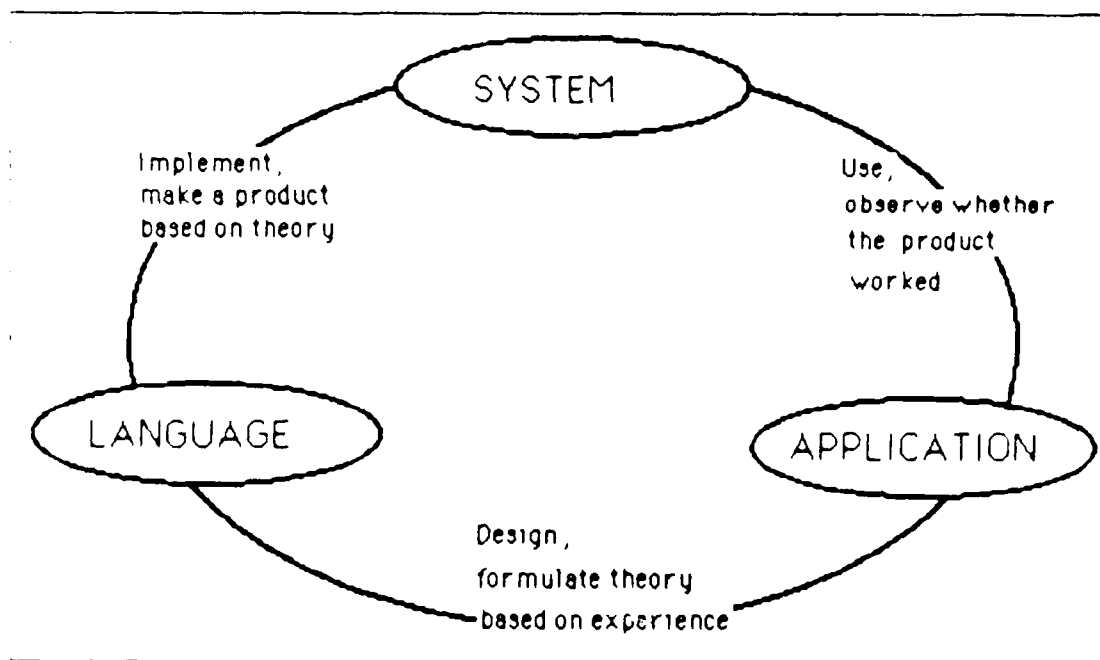


Figure 7.1 Incremental Development.

The program may evolve as a series of experiments, in which the result from one step gives the input to the design of the next step. During this process, the program may undergo drastic changes as the problem is better understood. The simple structure the programmer starts with grows by increasing the complexity of the modules. The enhancement process continues recursively until a finished product exists. The growth can occur both 'horizontally' through the addition of more facilities, and

'vertically' through a deepening of existing facilities and making them more powerful in some sense." [Ref. 46: p. 63]

In an ideal interactive integrated programming environment all of this process would take place within the computer system, using its resources (tools, hardware, etc.) to help. The user of this kind of system seems to be more of an "artist" than an operator. He/she will have the ideas, but use the system's resources to test and implement the program using incremental programming development.

MacLennan [Ref. 43] defines system development: "The entire process that takes an initial idea in the client's mind to a final constructed system that satisfies the client. That is, the entire 'life cycle' of the system, including later evolution to keep the client satisfied." This definition shows how dependent the, often computer illiterate, client is on flawless communication with the designer programmer in order to get what he/she wants.

C. HOW TO PUT THE USER IN CONTROL

The following three principles can help the average user of a computer system to feel in control of the computer resources:

1. Responsiveness, that is that the user's action at the computer should have direct visible results.
2. Permissiveness is to let the user, not the system, decide what to do next. The system should appear modeless to the user.
3. Consistency is to use the same interface for the whole environment, and for all applications in the environment.

Some producers of personal computers, like Apple's Macintosh and Commodore's Amiga, have applied these principles in their systems. They have to a large degree managed to put the user in control, but the cost has been much more complicated application software. The burden has been moved from the user to the hardware and the programmers of application software.

D. LISP IN INTERACTIVE PROGRAMMING ENVIRONMENTS

1. Why use Lisp

The basic syntax in Lisp is very simple, and the programs are naturally represented in simple Lisp data structures in a way that reflects the structure of the program. Lisp represents both data and programs with lists, therefore it is simple to write Lisp programs that read, preprocess, transform, and generate other Lisp programs. Lisp requires no declarations and therefore programs can be created incrementally, this would normally be difficult in a declarative language. The dynamic type system and flexible data structures make Lisp well suited for badly specified problems, and very well suited for experimentation. The interpreter in Lisp performs only one action, applying a function to its argument, therefore features (tools) like single stepping, tracing, and symbolic debuggers are easy to implement. In addition the simple syntax of Lisp makes logical presentation of code on a screen or on a page natural and easy. The compiler some Lisp implementations have, is there just to speed up the execution. The interpreter glues all the different tools together into an integrated system. It is never necessary for the designer programmer to think of his/her code as anything other than the source code. Such a view is in principle possible also in completely compiled programming languages, but it is much harder to achieve. The ease with which Lisp programs can manipulate other Lisp programs has given us a wide variety of Lisp programming tools. This library of programming tools evolved into a programming environment that supports all the phases of programming: design, coding, debugging, documentation, and maintenance. Interlisp is one example of the early programming environments developed around Lisp programming tools.

2. The Interlisp Programming Environment

a. *Introduction to Interlisp*

Interlisp is an interactive integrated programming environment based on Lisp. It is in extensive use, and has an extensive set of user facilities; including syntax extension, uniform error handling, automatic error correction (DWIM), an integrated structure based editor, a sophisticated debugger, a compiler, and a filing system [Ref. 47: p. 25-34]. The system is used at many sites, mostly at education centres (universities), and it is well documented and maintained. The Interlisp environment has evolved over time in an incremental fashion. Therefore the quality of the user interface has been, and still is, less than desired. The interfaces are inconsistent and

complex, and it is difficult to master all the tools and facilities. The unique thing about Interlisp is the following two attributes:

1. The high degree of integration.
2. How easy the facilities (tools) in the environment can be tailored, modified, and extended.

b. Some Facilities in Interlisp

The following section will discuss some of the important facilities in Interlisp. Many of the ideas are taken from Warren and Masinter: "The Interlisp Programming Environment" [Ref. 47].

The residential system is defined as a system where the primary copy of the program resides in the programming system as a data structure. The user makes changes to this copy during the interactive session, i.e. editing is done by modification to this data structure.

The file package is defined as a set of functions, and interfaces to other system facilities and tools. The user does not have to keep track of where things are, and which things have changed. In modern Interlisp the file package normally operates automatically, transparent to the user. The user no longer has to worry about maintaining his source files, but if he/she wants to make changes to this automatic bookkeeping it is easy to redefine or change these operations. The general file package supports the abstraction that the user manipulates his/her program as data while the file is just one of the possible representations of the code.

Masterscope is an interactive program for analyzing and cross referencing user programs in order to predict the effect of a proposed change to the program. Masterscope "determines which functions are called, how and where variables are bound, set, or referenced, which functions use particular record declaration, etc." [Ref. 47: p. 30] When Masterscope performs its analysis it builds a database of the result. The user has access to this database, and can interrogate using English like queries. In addition Masterscope can call the editor on all functions that contain expressions that satisfy certain relationships specified by the user. Masterscope adds another level of abstraction to the system because the user no longer has to remember what was changed, and therefore needs new analysis, but the system takes care of this. The interaction between different functions are done automatically, and transparent to the user.

Do What I Mean (DWIM) is the feature that fascinates me the most. The system invokes DWIM when it detects an error, then the DWIM attempts to guess what the user intended to do. DWIM is a collection of programs that makes reasonable interpretations when given unrecognized inputs at user level. The DWIM system is transparent to the user, and is an important part of the user interface. The simplest, and most visible, part of DWIM is the spelling corrector which attempts to find the closest match within a list of relevant items. This list is easily modified, so that the user can tailor it to his/her needs. The spelling corrector can be used to enforce standards etc, i.e. DWIM automatically transforms input to a standard syntax.

The Programmer's Assistant is an active intermediary between the user and the lower levels of the system. "The programmer's assistant records, in a data structure called the history list, the user's input, a description of the side effects of the operation, and the result of the operation" [Ref. 47: p. 32]. The "undo" command is closely related to the history list. As long as the user doesn't tell the system to do otherwise, the programmer's assistant will always be part of the user interface. In most cases the programmer's assistant is transparent to the user, and responds to commands that manipulate the history list. The history list keeps track of what the user has typed, so that keyboard input can be reused while just specifying what has changed. Interlisp records absolutely every change to the structure, but it does not record why it was changed.

c. Interlisp-D Programming Environment

Interlisp-D is in general Interlisp with windows added to it, and it is an example of a single user virtual memory. The user sees the interactive integrated programming environment as a collection of windows. Each window corresponds to a different tool, task or context. The introduction of bitmapped displays, pointing devices, and windows has greatly enhanced the user interface of the Interlisp-D system.

One of the major disadvantages with standard Interlisp is its cumbersome user interface, and Interlisp-D solves some of the problems.

E. AN OBJECT ORIENTED INTERACTIVE PROGRAMMING ENVIRONMENT

1. Why use Smalltalk

a. Introduction

The Software Concepts Group at Xerox Palo Alto Research Center had as the goal "to create a powerful information system, one in which the user can store, access and manipulate information so that the system can grow as the user's ideas grow. Both the number and kinds of system components should grow in proportion to the growth of the user's awareness of how to effectively use the system" [Ref. 30: p.VII]. The philosophy is to choose general principles and apply them uniformly. That means that if we have built some objects, we should always use enhance these existing objects when possible instead of creating new ones from scratch. The Smalltalk system lets the user make changes to the system itself while it is running, i.e. the user may crash the system by modifying some objects that are critical for the system.

The specification of Smalltalk-80's virtual machine describes the required behavior of any interpreter. An implementation of a Smalltalk-80 interpreter is only required to exhibit external behavior which is identical to that described by the formal specification as it appears in "Smalltalk-80: The Language and Its Implementation" by Goldberg and Robson. As long as the external behavior is preserved, the implementation can make design tradeoffs to increase the performance, or meet special needs.

The choice of a programming language to implement the Smalltalk-80 interpreter is based on the tradeoff between the performance needed, and the ease of implementation. The interpreter depends on an efficient mapping of the virtual machine architecture onto the available hardware resources of the processor. The resources include: registers, preferred memory locations, instruction sequences, etc. Generally a low level (assembly) language gives the implementor total freedom, but he/she must also take total responsibility for correct programming. This is in contrast to high level languages where the designer of the compiler did the general resource allocation (which often is not optimal for the Smalltalk interpreter).

b. Features in Smalltalk

The user interface consists of many facets and in the following discussion some of the more important ones are studied more in detail

Views are the rectangular areas on the display screen. Views may contain only text, only pictures, or a combination of the two. Views are the same as windows, i.e. to select a view is to enter a window.

The browser is a view of the classes in the Smalltalk-80 system. New classes are added to the system, and existing classes are examined and changed, using the browser. Programmers in Smalltalk-80 define classes and methods incrementally by editing in system browsers. To be able to share these class descriptions with others, files are used for communication. The files are called "code files" and allow the user to communicate source code between one Smalltalk system and another. The files can also be used to communicate information from the system to itself at a later point in time. The file that stores changes does so by appending to it, and therefore previous versions of the source code can always be found easily. This file also records several other kind of information in order to help recovery after a system crash: it marks execution of an expression in a code generator, occurrence of a snapshot, etc. The file format is used by the system to keep the source code for the methods on disk files, rather than within the memory of a resident system.

Error reporting is supplied with notifiers and debuggers. The process in which the error is encountered is suspended and a view of this process is created. The notifiers give a description of the process at the time the error was encountered. The debugger generally gives a more detailed view, but also allows the user to change the state of the suspended process before resuming it.

These interfaces along with the fact that Smalltalk lets the user change the system itself, so that it may crash, made it apparent that it was necessary to save to disk the entire state of the system at certain times. This is called a "snapshot" of the system, and is currently performed automatically from time to time. In addition the user can invoke the snapshot when needed. When a critical error occurs, the user "boots and resumes" his her work from the previous state saved in the last snapshot. In Smalltalk-80 the snapshot is represented by the virtual image format. The "changes" file is only altered by appending data to it, therefore any previous version can be found.

c. Smalltalk as a Programming Environment

The programming process is assisted by several classes in Smalltalk-80. Different classes are used to represent the user readable code and the machine executable form of methods. Objects are used to represent parsers, compilers, and

decompilers. Decompilers translate between the different representations (methods). Objects representing classes connect methods with the objects that use them. In addition objects representing organizational structures for classes and methods help the programmer keep track of the system's state, and objects representing histories of software changes help interface with other programmers. Finally objects also represent the execution state of a method; they are called contexts, and are similar to stack frames or activation records of other programming systems. Everything in Smalltalk is an object, and any object can be bound to any name because no names are typed. Smalltalk has dynamic type checking, like Lisp, while for example Pascal and Ada have static type checking. Dynamic type checking lets Smalltalk allow a message to be sent to an object only if that object has a method to respond to the message. Any object with the proper protocol may be passed to a method. In Smalltalk it is not possible to crash the system due to type violation.

The uniformity of Smalltalk is only valid within the system itself. It is not possible to maintain this uniformity in the interfaces to the external world, because the external world consists of disk files, printers, etc. that are not Smalltalk objects. All programs that want to share information with some other program meet this problem.

Smalltalk is a graphically oriented interactive integrated programming environment. The language is designed so that all components in the system that are accessible to the user can be presented in a meaningful way for manipulation and observation. Smalltalk builds on the model of independent communicating objects. Applications written in the language are viewed in the same way as the fundamental units from which the system itself is built. Interaction between the most primitive objects is viewed in the same way as high level interaction between the computer and the user. The pure object oriented programming languages are ideally suited for interactive integrated programming environments. A person working at a terminal responds to conditions and takes actions in time.

Most traditional systems are built around a kernel of code which cannot easily be modified. In the Smalltalk systems the kernel consists of machine code and microcode in order to implement a virtual machine. The kernel must be as small as possible to prevent the need for frequent changes to it. Smalltalk facilitates incremental design of the system, therefore we have the problem of ensuring the integrity of the system. The system tracer takes care of this problem. It is a program running inside of Smalltalk that copies the whole system out to a file while it is

running. The system tracer lets the user live in the system he/she is working on, without having to start from scratch every time the system crashes. One of the benefits from the system tracer is that it makes it easier to use a fully interactive integrated programming environment for production applications. The total system contains many tools facilities that are not needed in production systems, i.e. compiler, debugger, editor, communication, etc. The system tracer has the ability to strip off most of these unnecessary tools facilities. In addition the system tracer can be used to produce 'mutations' of existing programs. An example of this is how we can change the floating point number. We can include an appropriate transformation in the system tracer and write out a mutation of the old program, then we can replace the routine instruction in the virtual machine and start up again with a modified system. This modification method has been used in Smalltalk to change: floating point numbers, instruction set of the virtual machine, format of compiled methods, and encoding of small integers [Ref. 23; p. 26].

Some computer scientists feel that object oriented programming languages can take over some of the roles from the operating systems. Daniel Ingalls stated it even stronger: "An operating system is a collection of things that don't fit into a language. There shouldn't be one." [Ref. 48]

F. SUMMARY OF CHAPTER

The unique thing about interactive integrated programming environments are that they are used to represent and develop other programs, and to manipulate these other programs. The interactive integrated programming environment consists of a set of tools, and a tool is anything that can help in the programming process. The integration of tools are very hard, and today it is easier to do in an interpreted programming language with dynamic binding, like Lisp or Smalltalk than in a compiled programming language. In addition Lisp and Smalltalk facilitate incremental program development, i.e. the program evolves as a series of experiments in which the result from one step gives the input to the design of the next step.

The user interface may be enhanced if the following principles are followed: responsiveness, permissiveness, and consistency.

Interlisp and Smalltalk-80 are examples of interactive integrated programming environments that increase the programmer's capabilities. The integration of tools are so well done that the sum total is larger than the sum of the individual tools.

VIII. CONCLUSIONS AND RECOMMENDATIONS

A. CONCLUSIONS

The structure of our language, also programming languages and programming environments, define the boundaries of human thought. Therefore, even if the theoretical power of all languages are the same, programming languages and programming environments, at the moment, limit human creativity and ability to solve, new, large, and complex problems. Today's programming languages and programming environments are specialized, and not well suited for all types of problems.

Increased emphasis is put on the user interface. We have a lot of background knowledge in this area, but at the moment we don't know exactly how to produce the optimum user friendly interface. Bitmapped displays, menus and mouse-like devices have so far been the solution. The burden is moved from the user to the hardware and programmers of application software. The unique thing about interactive integrated programming environments are that they are used to represent and develop other programs, and to manipulate these other programs.

In general a homogenous representation of a programming language makes it easier to create an interactive integrated programming environment. Smalltalk and Lisp are examples of this. The languages are based on a relative small number of consistent abstractions, and seek to provide uniform treatment of different kinds of information: text, graphics, symbols, and numbers.

Object oriented programming languages do not give technical advantages, but cross a threshold of perception and make it easier for the human being to solve new and complex problems. To be a true object oriented programming language four features must be supported: information hiding, data abstraction, dynamic binding, and inheritance hierarchy. A result of these features is that this thesis supports the simulation paradigm as the most appropriate for object oriented languages. Smalltalk-80 is an example of this kind of language, but in addition Smalltalk-80 is an interactive integrated programming environment in itself.

Smalltalk-80 is both a programming language and an interactive integrated programming environment. The Smalltalk-80 system has the feature that it can modify itself and thereby produce a new interactive integrated programming environment adapted to the user's needs. It can simulate the new environment in the existing (old) environment, then use the simulation to actually produce the new system.

B. RECOMMENDATIONS

1. What Can be Done Now

In order to make the user feel more comfortable with his/her programming environment, new applications should build on skills the user already has instead of forcing him/her to learn new skills. The user should stay in control of the computer throughout the session. Emphasis should be put on responsiveness, permissiveness and consistency when the user interface is designed. A good naming convention can help the user to easily get the purpose of the functions, routines, objects, etc.

Interactive integrated programming environments based on interpreted programming languages like Lisp and Smalltalk seem to be best suited for environments producing new programs, and not so much for pure execution of computation heavy applications.

2. Future Research Areas

The interactive integrated programming environment of the future should pay more attention to the total project development problem. Therefore it should at least be developed in the following directions:

1. The design and programming tools should be even more integrated.
2. A good database system to keep track of versions etc. is needed.
3. The project management tools must be integrated with the rest of the system, including the database.
4. Tools to perform semantic analysis during programming is needed.
5. Reduce or eliminate the semantic differences between different languages environments. Consistent use of commands etc.

Both Lisp and Smalltalk let the user modify his/her own environment. Therefore we need to develop some well defined principles for the user interface in order to reduce the burden on the user even more. Detailed specifications of the underlying system is not needed.

APPENDIX A

SMALLTALK-80 TERMINOLOGY

This appendix contains a summary of Smalltalk-80's terminology used in this thesis. The definitions are taken from: "Smalltalk-80: The Language and its Implementation" by Adele Goldberg and David Robson [Ref. 30].

- **ABSTRACT-CLASS** A class that specifies protocol, but is not able to fully implement it, by convention, instances are not created of this kind of class.
- **CLASS** An object that describes the implementation of a set of similar objects.
- **INSTANCE** One of the objects described by a class; it has memory and responds to messages.
- **INSTANCE VARIABLE** A variable available to a single object for the entire lifetime of the object; instance variables can be named or indexed.
- **INTERFACES** The set of messages to which an object can respond. The only way to interact with an object is through its interface.
- **MESSAGE** A request for an object to carry out one of its operations.
- **MESSAGE SELECTOR** The name of the type of operation a message requests of its receiver.
- **METACLASS** The class of a class.
- **METHOD** A procedure describing how to perform one of an object's operations; it is made up of a message pattern, temporary variable declaration, and a sequence of expressions. A method is executed when a message matching its message pattern is sent to an instance of the class in which the method is found.

- OBJECT A component of the Smalltalk-80 system represented by some private memory and a set of operations.
- OVERRIDING Specifying a method in a subclass for the same message as a method in a superclass.
- PRIMITIVE An operation performed directly by the smalltalk-80 virtual machine.
- RECEIVER The object to which a message is sent.
- SUBCLASS A class that inherits variables and methods from an existing class.
- SUPERCLASS The class from which variables and methods are inherited.
- SYSTEM The set of classes that come with the Smalltalk-80 system.

AD-A184 127

A SURVEY OF OBJECT ORIENTED LANGUAGES IN PROGRAMMING
ENVIRONMENTS(U) NAVAL POSTGRADUATE SCHOOL MONTEREY CA
H HARKONSEN JUN 87

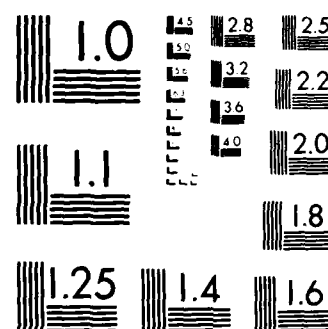
2/2

UNCLASSIFIED

F/G 12/5

NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

APPENDIX B

TOWER-OF-HANOI IN PROLOG

Tower-Of-Hanoi program in Advanced A.I. Systems' Prolog, Version M-1.13. In order to run the program you must specify the number of disks, and the names of the three poles used.

```
hanoi(0,From,To,Other) :- !.                % nothing to do.
hanoi(Disk,From,To,Other) :-
    Disk1 is Disk-1,
    hanoi(Disk1,From,Other,To),              % move top Disk-1 aside
    move(Disk,From,To),                      % move Disk
    hanoi(Disk1,Other,To,From).              % move top Disk-1 back

move(Disk,From,To) :-
    nl,write('Move disk '),write(K),
    write(' from pole '),write(From),
    write(' to pole '),write(To).

:- hanoi(3,pole1,pole2,pole3).
```

APPENDIX C

TOWER-OF-HANOI IN LISP

The Tower-Of-Hanoi program builds on a program found in "LISP" by Winston and Horn [Ref. 20: p. 88-90].

<pre> (defun Hanoi nil (Transfer '1 '3 '2 (read))) (defun MoveDisk (From To) (print (list From '-> To)) (terpri)) (defun Transfer (From To Using Height) (cond ((equal Height 1) (MoveDisk From To)) (t (Transfer from Using To (sub 1 Height)) (MoveDisk From To) (Transfer Using To From (sub 1 Height)))) </pre>	<pre> ;K disks on 1 first. ;Print instruction. ;Start new line. ;Transfer one disk. ;Move from From ;to Using ;using To as space ;(K - 1) disks. ;Move lowest disk. ;Move from Using ; to To ;using From as space ;(K - 1) disks. </pre>
--	--

APPENDIX D

TOWER-OF-HANOI IN PASCAL

The Tower-Of-Hanoi program is based on a program in "A Taste of Smalltalk" by Ted Kaehler and Dave Patterson [Ref. 49: p. 3-4]. MacPascal version 1.0 is used.

```
program TowersOfHanoi;  
  var  
    howMany : integer; (Number of disks)  
  procedure MoveTower (Height, FromPole, ToPole, UsingPole : integer);  
    procedure MoveDisk (FromPole, ToPole : integer);  
      begin  
        writeln('FromPole', FromPole : 3, ' ToPole', ToPole : 3);  
      end;  
    begin {MoveTower}  
      if height > 0 then  
        begin  
          MoveTower(Height - 1, FromPole, UsingPole, ToPole);  
          MoveDisk(FromPole, ToPole);  
          MoveTower(Height - 1, UsingPole, ToPole, FromPole);  
        end;  
      end; {MoveTower}  
    begin {TowersOfHanoi}  
      showtext;  
      writeln('How many disks do you want?');  
      readln(howMany);  
      MoveTower(howMany, 1, 3, 2);  
    end {TowersOfHanoi}
```

APPENDIX E

TOWER-OF-HANOI IN SMALLTALK-80

The Tower-Of-Hanoi program is taken from "A Taste of Smalltalk" by Ted Kachler and Dave Patterson [Ref. 49].

* METHOD MoveDisk:to:

moveDisk: fromPole to: toPole

"Move Disk from a pole to another pole.

Print results in the transcript window"

Transcript cr.

Transcript show: (fromPole

printString, '->', toPole printString).

* METHOD moveTower: from: to: using:

moveTower: height from: fromPole to: toPole using: usingPole

"Recursive procedure to move the disk at a height from one pole to another
using a third pin"

(height > 0) ifTrue: [

self moveTower: (height - 1) from:

fromPole to: usingPole using: toPole.

self moveDisk: fromPole to: toPole.

self moveTower: (height - 1) from:

usingPole to: toPole using: fromPole]

"Run the program by selecting and choosing 'do it' .

(Object new) moveTower: 3 from: 1 to: 3 using: 2"

LIST OF REFERENCES

1. Morrison, Phillip and Emily, eds., *Charles Babbage and His Calculating Engines*, Dover Publications, New York, 1961.
2. Whorf, Benjamin L., *Language, Thought, and Reality*, MIT Press, 1956.
3. MacLennan, Bruce J., *Principles of Programming Languages : Design, Evaluation and Implementation*, Holt, Rheinhardt and Winston, 1983.
4. Barstow, David R., Shrobe, Howard E., and Sandewall, Erik, *Interactive Programming Environments*, McGraw-Hill Book Company, 1984.
5. Miller, L. A., *Natural Language Programming: Styles, strategies, and contrasts*, IBM System Journal, Vol. 20, No. 2, 1981.
6. Ardir, Mark, *Tutorial Notes: Software Development Environments*, IEEE Computer Society, 1987.
7. Batimo, J., *Smalltalk with Alan Kay*, Information World, Vol. 6, No. 24, June 1984.
8. Brooks, Ruven, *Towards a theory of the cognitive process in computer programming*, Int.J.Man-Mach.Stud.9, 1977.
9. Weinberg, Gerald M., *The Psychology of Computer Programming*, Van Nostrand Reinhold Company, New York, 1971.
10. Dahl, Ole-Johan, Dijkstra, Edsger W., and Hoare, C. A. R., *Structured Programming*, Academic Press, New York, 1972.
11. Parnas, David L., *Software Aspects of Strategic Defense Systems*, Communication of the ACM, Vol. 28, No. 12, December 1985.
12. Uebbing, Johan and Young, Charles, *User Interface Performance Issues*, BYTE, August 1986.
13. Monk, Andrew, *Fundamentals of Human-Computer Interaction*, Academic Press, New York, 1985.

14. Brocks, Ruven, *A theoretical analysis of the role of documentation in the comprehension of computer programs*, Proceedings of Human Factors in Computer Systems, Gaithersburg, Maryland, 1982.
15. Kowalski, Robert, *Algorithm = Logic + Control*, Communications of the ACM, Vol. 22, No. 7, July 1979.
16. Booch, Grady, *Software Engineering with ADA*, Benjamin Cummings Publication Co., 1983.
17. Weinberg, Gerald M., *Rethinking Systems Analysis and Design*, Boston, Little, Brown, 1982.
18. Miller, George A., *The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information*, The Psychological Review, March 1956.
19. MacLennan, Bruce J., *A Simple Software Environment Based on Objects and Relations*, NPS52-85-005, Naval Postgraduate School, Monterey, California, 1985.
20. Winston, Patrick Henry and Horn, Berthold Klaus Paul, *Lisp*, Addison-Wesley Publishing Company, 1981.
21. Rowe, Neil C., *Artificial Intelligence*, Unpublished, Naval Postgraduate School, Monterey, California, 1986.
22. Buzzard, C. D. and Mudge, T. N., *Object-Based computing and the Ada Language*, IEEE Computer, March 1985.
23. Krasner, Glenn, *Smalltalk-80: Bits of History, Words of Advice*, Addison-Wesley Publishing Company, 1983.
24. Kay, Allan C., *Microelectronics and the personal computer*, Scientific American, September 1977.
25. Stefik, Mark and Bobrow, Daniel G., *Object-Oriented Programming: Themes and Variations*, AI Magazine, Vol. 6, No. 3, 1986.
26. Robson, David, *Object-Oriented Software Systems*, BYTE, August 1981.
27. Moon, David A., *Object-Oriented-Programming with Flavors*, Association for Computing Machinery, OOPSLA, 1986.
28. Apple, *MPW Pascal Language*, APDA#KMSPWP, Apple Programmer's and Developer's Association, 1986.

29. Nygaard, Kristen, *Basic Concepts in Object Oriented Programming*, ACM Sigplan Notices Vol. 21, No. 10, October 1986.
30. Goldberg, Adele and Robson, David, *Smalltalk-80: The Language and its Implementation*, Addison-Wesley Publishing Company, California, 1983.
31. MacLennan, Bruce J., *A View of Object-Oriented Programming*, NPS53-83-001, Naval Postgraduate School, Monterey, California, 1983.
32. Parnas, David L., *On the Criteria To Be Used in Decomposing Systems into Modules*, Association for Computing Machinery, 1972.
33. Davis, William S., *Systems Analysis and Design*, Addison-Wesley Publishing Company, 1983.
34. Bray, Gary and Pokrass, David, *Understanding Ada: A Software Engineering Approach*, John Wiley and Sons, New York, 1985.
35. Pascoe, Geoffrey, A., *Elements of Object-Oriented Programming*, BYTE, August 1986.
36. Webster, *Webster's Third New International Dictionary*, G. & C. Merriam Company, Massachusetts, 1961.
37. Strom, Rob, *A Comparison of the Object-Oriented and Process Paradigms*, ACM Sigplan Notices, Vol. 21, No. 10, October 1986.
38. Snyder, Alan, *CommonObjects: An Overview*, ACM Sigplan Notices, Vol. 21, No. 10, October 1986.
39. Snyder, Alan, *Encapsulation and Inheritance in Object-Oriented Programming Languages*, ACM Sigplan Notices, Vol. 21, No 11, November 1986.
40. Sandberg, David, *An Alternative to Subclassing*, Association for Computing Machinery, OOPSLA, 1986.
41. Agha, Gul, *An Overview of Actor Languages*, ACM Sigplan Notices, Vol. 21, No. 10, October 1986.
42. Brachman, Ronald J., *I Lied about the Trees*, AI Magazine, Fall, 1985.
43. MacLennan, Bruce J., *CS 4150 Lecture Notes*, Unpublished, Naval Postgraduate School, Monterey, California, 1987.
44. Burke, J., *Connections*, Little, Brown, Boston, 1978.

45. Khoshafian, Setrag N., *Object Identity*, Association for Computing Machinery, OOPSLA, 1986.
46. Sandewall, Erik, *The Lisp Experience*, Interactive Programming Environments, McGraw-Hill Book Company, 1984.
47. Teidelman, Warren and Masinter, Larry, *The Interlisp Programming Environment* IEEE, Computer, April 1981.
48. Ingals, Daniel, *Design Principles Behind Smalltalk*, BYTE, August 1981.
49. Kaehler, Ted and Patterson, Dave, *A Taste of Smalltalk*, W. W. Norton and Company, New York, 1986.

INITIAL DISTRIBUTION LIST

	No. Copies
1. Defense Technical Information Center Cameron Station Alexandria, VA 22304-6145	2
2. Library, Code 0142 Naval Postgraduate School Monterey, CA 93943-5002	2
3. Chief of Naval Operations Director, Information Systems (OP-945) Navy Department Washington, DC 20350-2000	1
4. Department Chairman, Code 52 Department of Computer Science Naval Postgraduate School Monterey, CA 93943	1
5. Professor Bruce J. MacLennan, Code 52M1 Department of Computer Science Naval Postgraduate School Monterey, CA 93943	1
6. Professor Gordon Bradley, Code 52Bz Department of Computer Science Naval Postgraduate School Monterey, CA 93943	1
7. Harald Haakonsen Hoegsetevegen 3B N-5047 Fana Norway	3

END

10-87

DTIC